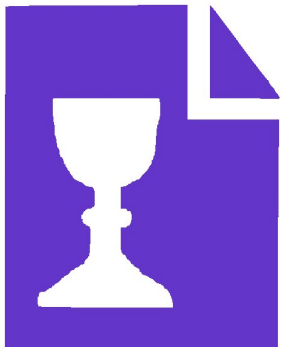
XXII. reál- és humántudományi Erdélyi Tudományos Diákköri Konferencia (ETDK)
Kolozsvár, 2020. május 21–24.

SoulMind

Lelkészek és gyülekezetek adatbázisa



Szerzők:

Cseke Alpár
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év
Király István
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Témavezetők:

dr. Simon Károly, egyetemi adjunktus
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar
Mátis Szilárd-Gábor, szoftverfejlesztő,
CodeSpring
Tüzes-Bölöni Kincső, szoftverfejlesztő,
CodeSpring

Kivonat

A SoulMind lelkészkataszter projekt célja lelkészek és gyülekezetek adatainak tárolása, megjelenítése, kezelése és felhasználása. Egy kényelmesen használható platform, amely leegyszerűsíti új adatok bevitelét, illetve a meglévő adatok szerkesztését. Különböző keresési kritériumok által lehetővé teszi az adatok átlátható böngészését és ezáltal könnyen használható felületet nyújt kutatói munka végzéséhez.

Mivel jelentős mennyiségű személyes adatot tárol, ezért a webalkalmazás zárt körű. Regisztráció-elfogadás alapú rendszert használ a felhasználói fiókok kezelésére; be nem jelentkezett felhasználók csak engedélyt kérhetnek a regisztráció által, míg egyre magasabb felhasználói szerepkörökkel egyre több funkcionalitás érhető el.

A rendszer az adatokat egy dokumentum alapú NoSQL adatbázisban tárolja, és egy Golangban megírt magas teljesítményű szerver biztosítja ezek kezelését, illetve a platform szolgáltatásait. A fejlesztés során nagy hangsúlyt kapott az oldal átláthatósága és használhatósága, az aktuális UX szabványoknak megfelelően.

Tartalomjegyzék

Bevezető

Mivel számos előre megírt és részletesen testreszabható, CRUD műveleteket (Create, Read, Update, Delete - a perzisztens adatok négy alapművelete [36]) támogató platform létezik, akár úgy is tűnhet, hogy nincs már szükség olyan új alkalmazások fejlesztésére, amelyeknek a célja egy bizonyos témakörhöz tartozó adatok kezelése és megjelenítése. Van viszont néhány szempont, amelyben a legtöbb általános adatkezelő platform hiányos. Ilyen például az adatok előfeldolgozása, a Bad Data [37] (rossz adat) kezelése, amely ha nincs megfelelően elvégezve, akkor a funkcionalitások rovására mehet.

A dolgozatban bemutatott alkalmazás többek között ennek a problémának a megoldására kínál lehetőséget egy adott doméniumban. Az esettanulmány tárgya egy webes felület, amely egy meglehetősen értékes és folyamatosan növekvő adathalmaz kezelését teszi lehetővé. A projekt kiindulópontjaként szolgáló erdélyi protestáns lelkészek gyűjteménye a Bad Data több karakterisztikájával is rendelkezik, mint a hiányzó mezők, a nem megfelelő adattípusok, a rossz mezőbe beírt adatok, a denormalizált és/vagy duplikált entitások. Az adatbázis mérete miatt viszont, amely ezernél több lelkészről rögzít közel 150 különböző adatot, a manuális tisztítás értelemszerűen nem egy lehetőség.

Ezeknek az életrajzi adatoknak a megjelenítésén és bővítésén kívül, egy második kulcsfontosságú motivációja is van a fejlesztésnek. A nyilvántartásban tárolt adatok egyedi információkat tartalmaznak, amelyek számos különböző témájú kutatás alapjául szolgálhatnak. A rendszer erre két külön megközelítésből is ajánl lehetőséget. Mindkét modul alapos szűrési módszerekre épül: bibliográfiai kutatás az egyéni lelkész profilok keresése és böngészése által, illetve statisztikai kutatás, amely a lelkészek szűrésének és az adatmegjelenítés finomhangolásának kombinációja által jön létre.

Ezenkívül még két másik fontos követelményt is meg kell említeni a szoftver bemutatásában. Az adatok személyes jellege miatt biztonságos elérésre van szükség. Ez egy többszintű felhasználói rendszer által jön létre, ahol már a legalsó szint is jóváhagyás alapú, és a magasabb jogosultsági körök egyre több funkcionalitást és adathozzáférést tesznek lehetővé. Ugyanakkor, mivel a weboldal célcsoportja magába foglal számítógépkezelésben kevésbé jártas személyeket is, a felhasználói felület megtervezése átfogó UX (User Experience) kutatás [39] eredménye.

A Lelkész Kataszter egy egyedi és nagyon specifikus megoldás egy amúgy viszonylag gyakori problémára. Sok különböző területen van szükség adatkezelő, -szűrő és -aggregáló platformokra. A bemutatott szoftver viszont, kínál több olyan funkcionalitást is, amelyek ezek közül egyikben sem jelennek meg, és amelyeknek helyettesítése több különböző külső szolgáltatást és magas szintű szaktudást igényelne. Ilyen sajátosságok például a dinamikus

ábrakészítő vagy a térképes keresés.

Az első fejezet tömören bevezeti az olvasót az applikáció fontosabb komponenseibe és az alapjául szolgáló architektúrába. A második rész összefoglalja a két fő építőelem implementálásához használt technológiákat. Ezután három különböző fejezet tárgyalja a megvalósítás részleteit. Ezt a fejlesztési eszközök és módszerek bemutatása követi, majd végül az alkalmazás használatának egy részletesebb leírása, felhasználói nézőpontból. A dolgozatot a következtetések levonása és a továbbfejlesztési lehetőségek ismertetése zárja.

A Codespring cég Mentroprogramjának keretén belül, nyári gyakorlatként indult az alkalmazás fejlesztése. Szakértő munkájukért köszönet illeti dr. Simon Károly és dr. Sulyok Csaba koordinátorokat, valamint Mátis Szilárd-Gábor és Tüzes-Bölöni Kincső mentorokat. A fejlesztés folytatódott a Csoportos projekt tantárgy keretein belül is, Béczi Eliézer, Jakab Szilárd, és Hunyadi Dániel hallgatók közreműködésével. Köszönet illeti a Kolozsvári Protestáns Teológiai Intézetet, ahonnan nem csak a weboldal ötlete és az alapjául szolgáló adathalmaz ered, hanem folyamatos személyes kapcsolattartással és tanácsadással is segítették a fejlesztést.

1. A SoulMind projekt

A fejezet bemutatja a SoulMind projektet, a szoftverplatform által kínált főbb funkcionalitásokat, illetve részletezi a rendszer felépítését.

1.1. A projekt funkcionalitásai

A platform feladata egy protestáns lelkészekről és gyülekezetekről szóló adatbázis elérhetővé tétele. A megszokott adatmanipulációs szolgáltatások mellett, mint az adatok megtekintése, új entitások felvezetése vagy a már létezőek szerkesztése, a weboldal tartalmaz több speciálisabb funkcionalitást is.

Az alkalmazás egy másik fókuszpontja a biztonság. Mivel a kezelt adatok egy része személyes jellegű, kiemelt fontosságú az elérés irányítása és korlátozása. Ez egy regisztráció-elfogadás alapú fiókmenedzsment rendszer segítségével valósul meg. Egy új felhasználói fiók létrehozása egy adminisztrátor jóváhagyásán múlik, aki az illető felhasználó elérési szintje felett is dönt.

Nem regisztrált vagy **be nem jelentkezett** felhasználók az autentikációs műveleteken kívül csak a főoldalt érhetik el. Itt a projekt leírása mellett a weboldalt érintő híreket olvashatják, utóbbiak fordított időrendben kerülnek megjelenítésre. Az autentikációs műveletek magukba foglalják a bejelentkezés oldalát, az elfelejtett jelszó cseréjére vonatkozó kérésnek a küldését, és a jelszócsere végrehajtásának felületeit, valamint a regisztrációs űrlapot, ahol név, e-mail, jelszó és egy rövid bemutatkozó szöveg megadásával lehet a hozzáférés-kérési folyamatot elindítani.

A bejelentkezett, **Vendég** szerepkörű felhasználók számára megnyílnak a keresőoldalak és az ábraszerkesztő is. Lelkészek keresésekor csak a találat kártyáin látható alapadatok érhetőek el, itt még nem lehetséges a teljes adatlaphoz navigálni. Megengedett viszont a gyülekezet profiloldalak megtekintése az összes adattal, amelyhez a lelkészek szolgálati listájából, illetve a gyülekezetek keresését támogató komponensen keresztül is eljuthatunk. Gyülekezetek keresése lehetséges térképen és listanézetben is.

A legalább **Olvasó** jogosultságú felhasználó egy új funkcionalitást érhet el, ez a lelkész profiloldalak böngészése. Ide a gyülekezeti adatlapok lelkészlistájáról és értelemszerűen a lelkészek keresését támogató oldalról navigálhat.

A **Szerkesztő** rangú felhasználók szerepköre két újabb funkcionalitás-csoporttal bővül. Létrehozhatnak új lelkész és gyülekezet adatlapokat néhány alapinformáció megadásával, amennyiben ezek még nem léteznek; egyező név és születési hely vagy egyező gyülekezetnév esetében a már létező entitás térül vissza válaszként. A profiloldalakon elérhető számukra a szerkesztés mód, amely az adatmezőket módosíthatóvá teszi és a képek és fájlok feltöltésének a lehetősége is megnyílik a szerkesztők számára. Továbbá, lelkészek esetében új családtagok,

1. ábra. Használati eset diagram az szoftverplatform fontosabb funkcionalitásaival és a felhasználói szintek hierarchiájával.

képzettségek, szolgálatok, foglalkozások stb. hozzáadása is lehetséges.

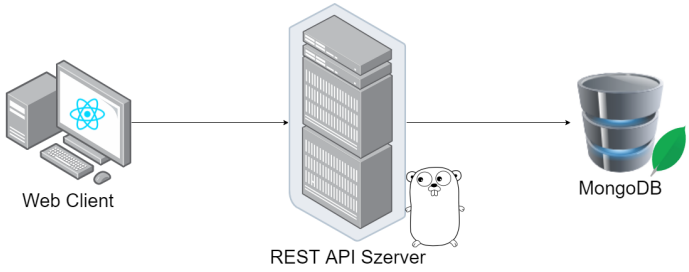
Az **Adminisztrátor** a legmagasabb felhasználói szint. Az összes eddigi funkció mellett, ők böngészhetik az oldal felhasználóinak adatait (név, e-mail, szerepkör), illetve törölhetik a létező fiókokat. Legfontosabb feladatuk viszont a regisztrációs kérések elbírálása, amely szintén az adminisztrációs felületen történik. Egy Adminisztrátor dönthet az új felhasználók elfogadásáról vagy visszautasításáról is, előbbi esetben a fiókhoz rendelt szerepkört is meg kell határoznia. Mindkét esetben írhat indoklást döntéséhez, amely szintén része a hozzáférést kérő felhasználónak küldött értesítő e-mailnek. Végül, de nem utolsó sorban, csak az ilyen szintű felhasználóknak van joguk a főoldalon új hírt létrehozni, a meglévőket pedig szerkeszteni vagy törölni.

Az utolsó felhasználói szint a **Tulajdonos**. Hatásköre megegyezik az Adminisztrátoréval, viszont csak egy létezik belőle, amely fiók már a webalkalmazás kitelepítéskor létezik és nem törölhető a többi Adminisztrátorral ellentétben.

Az elérhető funkcionalitások eloszlása a szerepkörök között részletesen végigkövethető az (1)-es ábrán lévő használati eset diagramon is.

1.2. Architektúra

A projekt architetktúrája három jól elkülöníthető komponensre osztható. A legalsó réteg a permanens adattárolást végzi, egy dokumentum-alapú (NoSQL) MongoDB adatbázisban [15]. Az adatbázis egy Golang-ben [12] megírt backend szerverrel áll kapcsolatban, ahol a lekért dokumentumok először a belső modell átfogó és teljes struktúráivá válnak, majd tovább lesznek



2. ábra. A projektben alkalmazott háromrétegű architektúra áttekintése.

alakítva különböző kompakt és célorientált adatátviteli objektumokká, a szerverhez beérkezett RESTful[1] API[2] kéréseknek megfelelően [16]. Ezek a kérések egy TypeScript nyelven [51] implementált, React [18] keretrendszerre épülő webes klienstől érkeznek.

A platform felépítése a Modell-Nézet-Vezérlő[3] programtervezési mintán alapszik [34]. Ez a megközelítés a (2)-es ábrán is végigkövethető. Mivel a Nézet réteg React-re épül, ezért a webalkalmazás SPA-ként (Single-page application) is kategorizálható, ami arra a tulajdonságára utal, hogy a nézet teljesen a web kliensen kerül felépítésre, az üzleti logika nagy része is itt van jelen [7], míg a backend szerver feladata nagyrészt csak az adatok kezelése és kiszolgáltatása.

[1] Representational State Transfer
[2] Application Programming Interface
[3] Model-View-Controller (MVC)

2. Technológiák ismertetése

Ez a fejezet tömören ismerteti a projekt backend és frontend részeinek implementálásához használt lényegesebb technológiákat, beleértve az adatelérési réteget is.

2.1. Szerver oldali technológiák

A dokumentum-alapú NoSQL adatbázisok, mint a projektben használt **MongoDB** is, az SQL-ből ismert sorok és oszlopok helyett úgynevezett dokumentumokat tartalmaznak. A MongoDB esetében a dokumentumok szerkezete hasonló a JSON[4] formátumhoz [14]. Kulcs-érték párokból áll, ahol az érték lehet skaláris, lista vagy akár egy beágyazott aldokumentum is. Ezek a hierarchikus struktúrára épülő entitások kollekciókban tárolódnak. Egy kollekción belüli dokumentumok szerkezete általában egymáshoz hasonló, de nem föltétlenül megegyező. Kollekciók közötti relációk létrehozása lehetséges az entitások egyedi azonosítói alapján, de ez nem mindig ajánlott. Ehelyett az egymáshoz kapcsolódó adatok egy kollekción és dokumentumon belül, aldokumentumokban tárolandóak. Fontos tulajdonsága az átfogó szűrési interfész, amely által többek közt tartomány- vagy reguláris kifejezés alapú keresések is végrehajthatóak.

Az adatbázist kezelő backend szerver **Golang**-ben van implementálva. A Google által fejlesztett és karbantartott, Golang-nek, vagy egyszerűen Go-nak is nevezett programozási nyelv legfőbb tulajdonságai az egyszerűsége és a teljesítménye. Stílusából fakadóan gyakran hasonlítják a C nyelvhez, mivel nem tartalmaz osztályokat és ehhez kapcsolódó lehetőségeket, mint például az öröklődés, hanem struktúrákon alapszik [33].

Kompilált nyelv, ezért futásidejű teljesítménye magas, és alkalmaz szigorú hibakeresést is, ugyanakkor van benne automatikus szemétgyűjtés (garbage collection). Az úgynevezett *goroutine*-ok pehelykönnyű és stabil többszálas futtatási lehetőséget biztosítanak, amit nagyon egyszerű előkészíteni. Egy másik fontos megkülönböztető eleme a nyelvnek, hogy próbál egyszerűen átlátható és ezáltal javítható, kezelhető maradni. Ennek elérése érdekében szakít a hagyományokkal, nem találkozunk benne öröklődéssel, osztályokkal, dinamikus típusokkal, de még a megszokott kivételkezelési *try-catch* sémákkal sem.

Annak ellenére, hogy viszonylag új nyelv, egyre nagyobb népszerűségnek örvend és rengeteg könyvtár érhető el hozzá, köszönhetően főként a Google kezdeti hozzáállásának [35]. Ezek a csomagok egy, a JavaScript npm-éhez[5] hasonló, *Go Modules* nevezetű csomagmenedzser által vannak kezelve, mind az automatikus letöltésük, mind a dinamikus verzióválasztás [25] szempontjából. A projekt implementálásához használt legfontosabb

[4] JavaScript Object Notation
[5] Node Package Manager

könyvtárak a Gin, a Viper és a mongo-go-driver.

A **Gin** egy HTTP alapú webes keretrendszer, amely szétosztja a beérkező REST hívásokat a hozzájuk párosított kezelő függvények között a kontrollerben [10]. A Gin részletesen konfigurálható különböző *middleware* rétegekkel, jelen esetben többek közt egy külön konfigurált naplózót, egy szesszióменedzsert, egy CORS konfigurációt [44] és egy szerep-alapú hozzáférést szabályozó interceptort tartalmaz. A maga egyszerűségével a Gin könyvtár meghatározza a backend szerver alapvető architektúráját, mivel a router a web klienssel történő összes kommunikáció alappillére.

A MongoDB készítői által fejlesztett **mongo-go-driver** könyvtár egy hídként szolgál a backend szerver és az adatbázis között [31]. Folyamatos kommunikációt hoz létre a két folyamat között és a Golang-ben, imperatívan megírt lekérdezéseket natív MongoDB parancsokká alakítja.

A **Logrus** könyvtár használatával vált lehetővé a hibák egyszerű visszakövetése. A *logger* (naplózó) színei és a megjelenő háttéradatok (időbélyeg, hiba helye) is beállíthatóak, illetve az elkészített konfiguráció könnyen integrálható más könyvtárak (pl. a Gin) belső loggereibe is [26].

A **Viper** egy konfiguráció-menedzser csomag [28]. A gazda operációs rendszer környezeti változóiban, kódban, különböző kulcs-érték típusú konfigurációs fájlokban, vagy a konténerizált kitelepítés paraméterei között megadott konfigurációs változókat gyűjti össze és priorizálja, meghatározva a végső értékeket. Ez nagyon fontos a backend szerver számára, hiszen több különböző környezetben is futhat, változó konfigurációval, legyen szó lokális tesztelésről, kitelepítésről, folyamatos integrációról vagy konténerizált lokális futtatásról.

2.2. Kliens oldali technológiák

A Facebook által eleinte belsőleg fejlesztett, majd 2013-ban nyilvánossá tett **React** könyvtár jelenleg az egyik legnépszerűbb JavaScript-es keretrendszer MVC alapú webes applikációk fejlesztéséhez. Ezt a JavaScript hivatalos csomagkezelője, az npm letöltési statisztikái is igazolják [40]. Más frontend könyvtárakkal szembeni előnyeinek megértéséhez több fogalom is tisztázásra szorul.

Komponens-orientált keretrendszer, a felhasználói felület elemeiből különálló és egyedi komponensek hozhatóak létre, ami nemcsak a kód strukturálhatósága és könnyebb szerkeszthetősége miatt hasznos, hanem mert ezek a komponensek a kívánt mennyiségben újrahasznosíthatóak a webalkalmazás különböző részeiben, de akár más projektekben is.

Az opcionális, bár a fejlesztők által javasolt JSX[6] szintaxis is egyedi vonás. Az imperatívan,

[6] JavaScript XML

```
class Hello extends React.Component {
  render() {
    return React.createElement(
      'div',
      null,
      `Hello ${this.props.toWhat}`
    );
  }
}

ReactDOM.render(
  React.createElement(
      Hello,
      {toWhat: 'World'},
      null
  ),
  document.getElementById('root')
);
```

(a) Alap JavaScript szintaxis.

```
class Hello extends React.Component {
  render() {
    return(
      <div>Hello {this.props.toWhat}</div>
    );
  }
}

ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root')
);
```

(b) JSX szintaxis.

1. kódrészlet. A React két választható szintaxisa.

sima JavaScript-ben történő fejlesztés helyett az elemek HTML-szerűen ágyazhatóak egymásba. A háttérben ez a React osztályok konkstruktorainak a *"<></>" shorthand* segítségével történő meghívásán alapszik, ez teszi lehetővé, hogy az (1)-es kódrészlet[7] bal oldalán használt szintaxis helyett a jobb oldalon látható módon deklaráljuk és példányosítsuk a komponenseket, amely nemcsak rövidebb de sokkal közelebb is áll a HTML alapú gondolkodásmódhoz. Fordításkor a JSX-el írt kód is az (1a) ábrán látható alakot veszi fel, így teljesítménybeli hátránya nincs.

A React használata az egyirányú adatáramlás köré szerveződik. Az adat a felsőbb komponensekből *immutable* változókon keresztül adódik át a belső rétegeknek, úgymond "lefelé folyik". Ezek a komponensek viszont nem képesek közvetlenül változtatni a kapott adatokat, kizárólag csak a szintén lefele átadott függvényeken keresztül módosíthatják azokat, amely függvény visszahívásokat idéz elő, így az akciók "felfelé folynak", az adatot birtokló komponens felé.

Egy másik sajátosság a *state* fogalma. Ez azon alapszik, hogy a szükséges adatok és változók a tőlük függő komponensek részei, mégpedig úgy, hogy minden ilyen erőforrás a virtuális DOM (Document Object Model) lehető legalacsonyabb szintjén van, ahonnan még az összes érintett alkomponens elérheti az egyirányú adatfolyam által. Egy másik fontos tulajdonsága a *state*-nek, hogy a hozzárendelt változók értékei figyelve vannak és változásuk esetén a komponens érintett részei frissülnek.

A virtuális DOM koncepciója is React-specifikus. Lényege, hogy a böngésző statikus

[7]Forrás: https://reactjs.org/docs/react-without-jsx.html, utolsó megtekintés dátuma: 2020-04-30

DOM-ján kívül a JavaScript motor is tárolja az oldal elemeinek fa-struktúráját és az előbbit érintő események hatására a VDOM a Diffing-algoritmus alapján csak az esemény által érintett komponenseknek a megváltozó részeit frissíti [20], a teljes komponens vagy akár teljes weboldal helyett.

A **TypeScript** nyelv lényegében a JavaScript típusossá bővítése, a kompilálás után arra fordítódik. Mivel úgynevezett *supersetje* a JavaScript-nek, és így bármilyen JavaScript kód egyben helyes TypeScript kód is, ezért a bevezetett szintaktikai elemei opcionálisak. Ezek között van a típus-annotációk megjelenése, és az ebből adódó, fordítási időben történő típusellenőrzés, a komponensek és függvények bemeneti értékeinek meghatározását segítő interfészek használatának a lehetősége, vagy a generikus típusok deklarálása is.

A **Semantic UI** egy CSS keretrendszer weboldalak témázásához. A könyvtár React-kompatibilis változata a CSS osztálynevek statikus használata helyett React-ben megírt komponenseket használ [23], ami lényegesen könnyít a fejlesztésen, mivel a különböző megjelenítési opciókhoz rendelt értékek a TypeScript által típusosak és statikus ellenőrzésre kerülnek.

Az **Axios** könyvtár felelős a backend szerverre küldött HTTP kérésekért. A könyvtár alkalmazza a *promise*-okra épülő aszinkron rendszert [30], ami annyit tesz, hogy a kérések után a program nem áll meg a választ várva, hanem tovább halad és majd az esetlegesen beérkező válasz vagy hiba interceptorok használatával félbeszakítja a programot az adatok átadásához. Másik fő előnye, hogy a könyvtár integrálódik a TypeScript opcionális szabályaihoz, ezáltal a kérést küldő függvényei generikusok, és így a visszatérített JSON struktúrák automatikusan a megfelelő interfésszé konvertálódnak.

A gyülekezetek térképe a **Leaflet** segítségével jeleníthető meg [5]. Ez az ingyenes OpenStreetMap[8] API-ról kéri le a jelenlegi térkép összerakásához szükséges *tileset-nek* nevezett darabokat. Ehhez társul a React-Leaflet-MarkerCluster kiegészítő csomag, amely a térképre helyezett egymáshoz közeli marker-ek csoportosítását, illetve az azonos koordinátákra helyezett marker-ek (több gyülekezet ugyanabban a városban) kattintásra történő szétszórását oldja meg.

[8] `https://www.openstreetmap.org/`

3. Az adatelérés megvalósítása

A fejezet az adatok tárolásának és elérésének, illetve az adatfeldolgozás lépéseinek leírását tartalmazza.

3.1. A dokumentum-alapú adatmodell

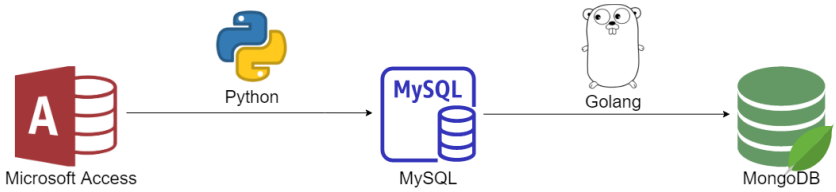
A projekt alapjául és motivációjául szolgáló adatbázis a kezdetben Accessben[9] volt, több száz különböző mezővel melyek változó mennyiségben és változó adattípusokkal voltak feltöltve, így jelentős különbség volt az entitások között az elérhető adatmezők mennyiségének és minőségének szempontjából.

A MongoDB által alkalmazott dokumentum-alapú adatmodell egyik olyan tulajdonsága, amely szorosan illeszkedik a projekt elvárásaihoz, hogy nem szükséges az adatmezőket és azok típusait rögzítő séma. Így az összes rendelkezésre álló adatot változtatás nélkül meg lehetett tartani és fel lehetett használni, még ha az adott mező csak nagyon ritkán vesz fel értéket, vagy ha ez az érték típusa nem illeszkedik az elvárthoz. Mindezt úgy, hogy nincs feláldozva sem a teljesítmény sem az átláthatóság a rengeteg hiányzó és/vagy hibás érték miatt. A NoSQL adattárolási megoldások ebben a tulajdonságukban különböznek a tradicionálisabb SQL adatbázisoktól: szigorúságot áldoznak fel rugalmasságért [6].

A projekt esetében öt különböző kollekció van definiálva, ezek tartalma és felhasználása a következő:

- A *Priests* gyűjtemény tartalmazza a lelkészekkel kapcsolatos információkat. Itt potenciálisan akár 150 különböző adatmezőről is szó lehet, melyek kategorizálva a következők: általános adatok, család, képzettségek, szolgálatok, egyházi és világi tisztségek, irodalmi munkásság, fegyelmi ügyek, képek és fájlok.
- A *Parishes* kollekcióban kerülnek tárolásra a gyülekezetekkel kapcsolatos adatok, mint a gyülekezet neve (illetve régi nevei), történelme, koordinátái, a település és az itt szolgált lelkészek listája.
- A *Users* tárolóban a rendszer felhasználói találhatóak. Ehhez tartozik a felhasználó neve, e-mail címe, hashelt jelszava és hozzáférési szintje.
- A jelszóváltoztatás sikerességéhez szükséges tokeneket a *NewPasswordTokens* gyűjtemény tartalmazza. Ehhez tartozik egy lejárati időbélyeg is, mely által a token az új jelszó igénylése után csak 2 napig érvényesíthető.
- Végül, az *Articles* kollekció tartalmazza a főoldalon bárki által megtekinthető híreket, amely címből, tartalomból és a keletkezés dátumából tevődik össze.

[9] Az Access a Microsoft Office csomag grafikus felületű adatbázis-kezelő rendszere



3. ábra. Az adatfeldolgozás lépései és technológiái.

3.2. Az adatfeldolgozás lépései

Az előző alfejezet elején említett okok miatt szükséges volt az adatok előfeldolgozása, majd átalakítása egy használható formába.

Első lépésként, egy Python[10] szkript segítségével a lelkészeket tartalmazó Access adatbázis egy relációs adatbázissá lett konvertálva, ez segített az adatok egymáshoz kapcsolódásának megértésében. Ezután következett a Golang-ben írt szkriptek segítségével a MongoDB-s kollekciók felépítése. Az eddigi folyamat technológiáit a (3)-as ábra is végigköveti.

Mivel az eredeti adatbázis csak lelkészeket tartalmazott, a gyülekezetek az egyéni lelkészek kihelyezései alapján lett egy algoritmus által összesítve és külön gyűjteményben eltárolva. Így a produkciós adatbázisban a gyülekezeteknél csak a gyülekezet neve és az ott szolgált lelkészek és segédlelkészek listája érhető el. A többi mező, mint a név eredete vagy a gyülekezet történelme üres, a protestáns intézet könyvtárosai által lesz majd feltöltve. Szintén a lelkészek adataiból való generálásnak a következménye az is, hogy néhol hasonló néven több gyülekezet is szerepel. Ezt nagyrészt elgépelések okozták és az egyértelmű hibákat a csapat próbálta algoritmikailag kiszűrni, viszont nem alkalmazott összetettebb szűrést az adatvesztés elkerülése érdekében. Nem rendelkezve a kellő szaktudással az adott területen, a csapat nem végezhetett manuális javítást sem. Sok esetben két gyülekezet/falu neve tényleg csak egy betűben különbözik, a nagyobb városok esetében az idők folyamán több gyülekezet is létrejött, illetve a lelkész adatai között a gyülekezet nevét tároló mező tartalmazhat információkat az ott szolgáló lelkész munkájáról is. Így a megegyező gyülekezetek adatainak egybeolvasztása is az alkalmazás felhasználóinak feladata lesz. Hasonló érvek alapján, ugyanez érvényes a lelkészek személyes adatainak esetében is.

A gyülekezetek nagy részéhez tartoznak koordináták, amelyek a gyülekezet neve alapján a Google Places API-jából lettek összegyűjtve [13] és beillesztve két Pythonban írt szkript segítségével. A megegyező koordinátákra mutató, hasonló nevű gyülekezetek térképen való megjelenítésének megoldása a web kliens technológiáiról szóló 2.2. alfejezetben már kifejtésre került.

[10] A Python egy magasszintű, interpretált programozási nyelv

4. Szerver oldali megvalósítások

A fejezet a backend szerver részletes felépítését taglalja, bevezetve az olvasót néhány implementációs döntés hátterébe.

4.1. Kapcsolódás az adatbázisokhoz

Az adatelérési réteg egy sor általános interfész köré van szervezve. A *Store* interfész mutatja, hogy egy implementáció milyen különböző *repository struct-okat* kell tudjon visszatéríteni, majd ezek közül mindegyikhez tartozik még egy-egy interfész, ami rögzíti a repository függvényeinek fejléceit, azaz nevét, illetve bemeneti és kimeneti változóinak típusát.

Jelenleg ezeknek csak MongoDB-s implementációja létezik, de a lehetőség adott az egyszerű bővítésre vagy akár adatbázis-váltásra is, mivel a controllerek nem tartalmaznak MongoDB-specifikus kódot. Ezen törekvés bemutatására példa a lelkészek szűrésének esete, ahol a mezők értékei az API kérés URI-jében érkeznek be a controllerbe, &-el elválasztott kulcs-érték párok listájaként. A MongoDB-s lekérés a szűrés mezőit egy dokumentum struktúra szerint várja el, de a controllerben mégsem egyből ez hozódik létre, hanem a kulcs-érték párosok egy általános *dictionary* adattípusba kerülnek, amit majd a repository függvénye feldolgoz és helyben hozza létre a speciális szintaxist tartalmazó natív BSON (Binary JSON) struktúrát, hogy létrejöhessenek a reguláris kifejezésekre és tartomány-korlátolt értékekre alapuló keresési feltételek.

A produkciós adatbázisnak az előző fejezetben indokolt gyakori újragenerálása miatt a fejlesztési környezetben a backend egy MySQL adatbázissal is kommunikál. Ezek az importáló függvények csak a szerver indulásakor futnak és a MongoDB-t kezelő interfész *deleteAll()* és *insertMany()* függvényeit használják a *batch* kérések végrehajtásához. A MySQL adatbázis nem része a végterméknek, csak fejlesztés közben volt használva.

Ezen kívül, a repository függvényei teljesen általános felépítésűek és a fejlesztés hasonlítható a hangsúlyosabban objektumorientált nyelvekben elérhető ORM[11] rendszerek konfigurálásához az SQL adatbázisok esetében.

4.2. Data Transfer Objectek

A DTO-k vagy Data Transfer Object-ek olyan köztes entitások, amelyek jelen esetben a backend részen vannak létrehozva az adatbázisból kiolvasott entitások belső modellje alapján, hogy az API által a web kliensnek JSON-ként legyenek küldve, vagy a web klienstől érkeznek, hogy a backend szerver feldolgozza őket. A DTO-k használata szorosan összefügg

[11] Object Relational Mapping

az assemblerek fogalmával. Ezek olyan segédfüggvények, amelyek a DTO-kat alakítják a belső modell típusaivá, vagy fordítva.

A programtervezési minta haszna abban rejlik, hogy a webalkalmazás különböző részein az adatbázisban tárolt entitások különböző mezőire van szükség [27]. Például, keresési találatok megjelenítésekor a lelkészekhez rendelt információk csak egy kis (kb. 10%) része kell: a fontosabb biográfiai dátumok és a szolgálatok helyei. A profiloldal megjelenítéséhez viszont szükség van az összesre, kivéve néhány belső azonosítót amit csak a backend használ. Az ábrakészítő megint más részhalmazát használja az adatoknak, teljes dátumok helyett csak éveket és összetett struktúrák helyett legtöbbször csak azok számosságát. Ezek az adatok mind ugyanabból az adatbázisban tárolt entitásból erednek, viszont különböző DTO-kká lesznek alakítva és kiküldve API hívástól függően, tehát a web kliens számára ezek mind teljesen különböző entitások.

A mező-szelekció mellett típuskonverzió is történik, a backend bizonyos adattípusai, mint pl. az adatbázis-specifikus azonosítók, JSON kompatibilis alakot kapnak. Ez utóbbi nem az assemblerek által történik, hanem a Golang struktúrák annotációi alapján jöhet létre. Például, a koordináta modellben a

```
Lat float64 `json:"lat,string,omitempty" bson:"lat,omitempty"`
```

sor azt jelenti, hogy a szélességi kört jelölő *Lat* mező lebegőpontos típusú az adatbázisban és a backenden, viszont amint JSON-ná alakul, karakterláncként lesz kiküldve a web kliens számára. Az *omitempty* jelölés miatt pedig, ha a mezőnek nincs értéke, akkor nem jelenik meg üres karakterláncként sem az üzenetben, sem az adatbázisba való mentéskor.

A DTO-k használatának több előnye is van. Jelentősen csökken a HTTP üzenetek mérete, ezáltal gyorsabbá és megbízhatóbbá válik a kommunikáció. Ugyanakkor, egyszerűsíti a web kliens fejlesztését is, mivel az adott komponens fejlesztője csak a megjelenítendő mezőket érheti el, és mindegyiket használnia is kell valahol. A DTO-k használatának biztonságot érintő aspektusai a 4.4. alfejezetben kerülnek tárgyalásra.

4.3. RESTful API Ginnel

A controller réteg fájljai olyan függvények gyűjteményéből állnak, amelyek a *router* által a REST API valamelyik végpontjához lettek rendelve, hogy az oda beérkező hívásokat feldolgozzák.

A (2)-es kódrészletben megfigyelhető, ahogy egyetlen függvényhívás által hozzátársul az azonosítót is tartalmazó *"/api/priests/:id/files"* útvonal a *priestCtrl* controller struktúra egy specifikus függvényéhez, mindezt csak bejelentkezett, Szerkesztő vagy magasabb rangú felhasználók számára téve elérhetővé.

```
priests := myRouter.router.Group("/api")
{
    priests.POST(
        "/priests/:id/files",
        auth.Authentication(auth.EDITOR, auth.ADMIN, auth.OWNER),
        myRouter.priestCtrl.UploadFile
    )
    ...
}
```

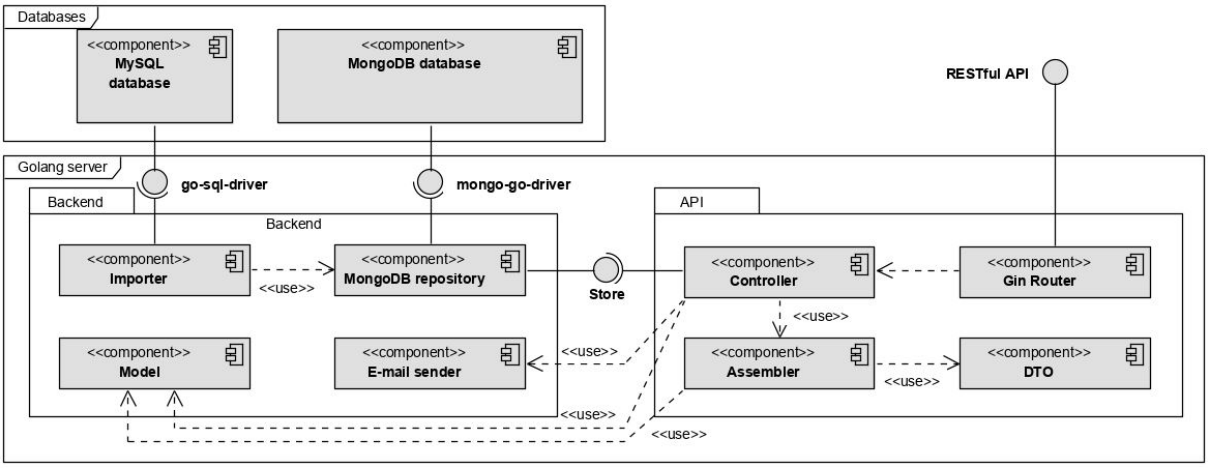
2. kódrészlet. A *router* használata.

A controller függvényei referenciát kapnak egy *gin.Context* típusú változóra, amelyből többek közt kinyerhetik a HTTP hívás paramétereit vagy a híváshoz csatolt adatokat. Az adatok kinyerése JSON dekóder segítségével történik, amely által a JSON üzenetből DTO-k vagy egyből a belső modellhez tartozó entitások lesznek. A dekóder ugyanakkor megoldást nyújt az érvénytelen üzenetek ellen is, lehetőség van az ismeretlen mezőket tartalmazó üzenetek automatikus elutasítására. Dekódolás után a controllerben adatbázis műveletek meghívása történik a kapott adatok alapján. Ezt rendszerint assemblerek használata követi vagy előzi meg.

Ezek a függvények a sikeres lefutás vagy bármilyen hiba esetén ugyanúgy egy választ térítenek vissza a web kliens számára. Ez a válasz állhat csupán egy szimpla HTTP státuszüzenetből (200 - "Oké", 204 - "Nincs tartalom", 400 - "Hibás kérés", 500 - "Szerverhiba" stb.) vagy tartalmazhat adatokat is, mint például egy entitáslista egy GET kérés esetében, vagy az *insertedID* mező az adatbázisba újonnan beszúrt entitás azonosítójával egy POST kérés esetében.

Az applikácó SPA tulajdonságából adódóan az alkalmazás-logika nagy része a web kliensen hajtódik végre, a backend szerveren csak adatbázist érintő műveletek történnek. Ezért itt, a megszokottal ellentétben, nincs egy külön szerviz réteg az adatbázist kezelő függvények és a controller között, hanem közvetlen a kapcsolat. A controllerben történnek az olyan műveletek, mint regisztrálás vagy bejelentkezés esetén az e-mail címhez rendelt fiók (nem) létezésének ellenőrzése, jelszó módosításakor a token érvényességének ellenőrzése, a jelszavak titkosítása és összehasonlítása, az értesítő e-mailek kiküldése, vagy egy lelkész kihelyezéseinek szerkesztésekor az érintett gyülekezetek módosítása is. Ezek miatt néhány controllernek több repository struktúrához is van hozzáférése.

Az eddigi három alfejezet által részletezett architektúra a (4)-es komponens diagramon is végigkövethető.



4. ábra. A backend szerver komponens diagramja.

4.4. Biztonság és autorizáció

A biztonság askpektusa a fejlesztés folyamán végig prioritás volt. A web kliensen alkalmazott, 5.3. alfejezetben tárgyalt mechanizmusok mellett a backend szerver is több különböző módszerrel igyekszik védeni az adatokat.

Ilyen megoldás az előző alfejezetben a (2)-es kódrészlet alapján már tárgyalt interceptor is, amely a híváshoz rendelve elsősorban a be nem jelentkezett küldőket szűri ki, az opcionális paramétereinek segítségével pedig tovább szigoríthatja az elérés feltételeit a felhasználói szint szerint. Mindez a HTTP kéréshez csatolt süti alapján történik, az ebben található szesszió azonosító alapján ellenőrizhető a bejelentkezés, és a bejelentkezéskor a szerver oldali szesszióhoz rendelt változók értéke is.

A 4.1. alfejezetben tárgyalt DTO-k egy újabb réteg biztonságot kölcsönöznek a platform számára. Használatukkal, a kereséshez hozzáférő, Vendég szintű felhasználó trükközés által sem érheti el a lelkészek olyan biográfiai adatait, amelyek a keresési találatok megjelenítésénél nincsenek használva és a felhasználói szintje számára így nem elérhetőek. Ez a mező-szelekció a DTO-k számára magába foglal olyan adatokat is, mint különböző belső azonosítók és időbélyegek, melyeknek a web klienshez való küldése szintén potenciális biztonsági rést jelenthetne.

A jelszavak védelme több szintre is bontható. Szintén a DTO-knak köszönhetően a felhasználók jelszavai semmilyen körülmények között nem hagyhatják el a backend szervert. Továbbá, mivel a kitelepített alkalmazás konténerizált és a külvilággal kommunikáló *proxy szerver* csak a web klienset és a backend szervert éri el, ezért az adatbázishoz történő illetéktelen hozzáférés teljesen lehetetlen. Ezen felül, regisztrációkor a jelszavak titkosítására és többszörös sózására kerül sor az alapkönyvtár részét képező *bcrypt* segítségével.

4.5. Automatikus e-mail értesítések

Egy, a projekt számára készített Gmail fiókot használ az alkalmazás, és a szintén Google által biztosított ingyenes SMTP[12] szerver segítségével vannak kiküldve az értesítő e-mailek [1]. A szolgáltatásnak van egy napi maximum 500 elküldött e-mailig terjedő korlátozása ingyenes felhasználók számára, viszont a leendő felhasználói bázist és annak a használati frekvenciáját becsülve ez mindenképp elégséges egy ideig.

Ha egy új hozzáférési kérés érkezik be, az összes adminisztrátor értesítő e-mailt kap, amely tartalmazza a csatlakozni vágyó felhasználó adatait és motivációs üzenetét. Elfogadás vagy elutasítás esetén az illető személy e-mailt kap fiókja sorsáról, mely szintén tartalmazza a döntéshozó adminisztrátor indoklását is. E-mail küldésre ezen kívül még jelszómódosításkor kerül sor, amikor a felhasználó egy 24 órás lejáratú aktiváló tokent tartalmazó linket kap a jelszómódosítás folytatásához.

A dinamikus adatok, mint például a felhasználó neve vagy a közvetített üzenet, a Golang alapkönyvtárát képező *html/template* csomag segítségével illesztődnek be az előre megírt és formázott HTML e-mail sablonok szerkezetébe.

Megemlítendő még, hogy mivel bármilyen külső szolgáltatástól való függés bizonytalansággal járhat (SMTP nélkül viszont valószínűleg spamként lennének jelölve az e-mailek), ezért a backend szerver folyamatosan követi az e-mail küldő szolgáltatás állapotát, az SMTP szerver által utoljára visszatérített érték alapján. Így valahányszor hiba történik, amíg a rendszernek nem sikerül újra e-maileket kiküldenie, addig az adminisztrátori oldal tetején egy figyelmeztetés látható, hogy értesíthessenek egy karbantartót a hibáról.

[12] Simple Mail Transfer Protocol

5. Kliens oldali megvalósítások

A React-hez hasonló SPA-orientált frontend keretrendszerek megjelenése óta a fejlesztési komplexitás egyre inkább a web kliensre helyeződik [3]. A fejezet a projekt fejlesztése során használt megoldásokat tárgyalja.

5.1. React állapotmenedzsment Store nélkül

A 2.2 alfejezetben tárgyalt *react state* nagyobb alkalmazások esetében gyakran az adatok komponensek közötti átláthatatlan szétszórásához vezetett. Ennek a hatására a React megjelenését gyorsan követték olyan könyvtárak is, amelyek ezt a program state-et gyűjtik össze egy vagy több úgynevezett *store*-ba. Az ilyen könyvtárak, mint a *Redux* vagy a *MobX*, követhetőbbé tették a state-et, viszont összetett szintaxisuk miatt megnövelték a kód méretét és bonyolultságát [29].

Egy másik probléma abból fakad, hogy a React komponensek működése különböző életciklus-metódusoktól függ, amelyek egy helyre gyűjtik az egy történés által kiváltandó akciókat. Ilyen függvények pl. a *componentDidUpdate()*, amely frissítés után hívódik meg, a *componentDidMount()* ami DOM-ba illesztéskor fut le, vagy a *componentWillUnMount()* amely a komponens eltüntetésekor aktiválódik. Ez a logika az egy elemhez kötődő különböző történések több függvény közötti szétszórásához vezet, ami a komplexebb komponensek esetében megnehezíti az elemek szerkesztését.

Ezekre próbál megoldást kínálni a *Hooks API*. Ez a frissítés a projekt kezdete előtt kevéssel jelent meg, így a hivatalos dokumentáción [19] kívül kevés külső erőforrás [41] állt rendelkezésre.

A hookok olyan függvények, amelyek által a komponens a *react state*-be “kapaszkodhat”, hogy a komponens elemeihez rendelt adatok változására reagáljon az oldal. Fontos megemlíteni, hogy csak funkció-alapú komponensekben használhatóak, így teljes mértékben helyettesítik a fentebb tárgyalt, életciklushoz kötődő függvényeket, amelyek csak osztályokon belül érhetőek el. Egyik ilyen fontosabb hook a *useState()*, amely egy változót és az azt változtatni képes függvényt a state-hez csatolja. A

```
const [filters, setFilters] = useState(initFilters);
```

sor például a szűrők értékeit köti a state-hez, az üres mezőket tartalmazó *initFilters* objektumot megadva kezdeti értékként. Egy másik gyakran használt hook, amely egymagában képes a legtöbb életciklus-függvény egyesítésére a *useEffect()*. A (3)-as kódrészlet végén lévő zárójelben megadott változó módosulásának hatására hívódik meg a belső függvény, ami ebben az esetben a *response* objektum létrejöttének ellenőrzése után (ami aszinkron módon érkezik

```
useEffect(() => {
    if (response) {
        setShowLoader(false);
    }
}, [response]);
```

3. kódrészlet. A *useEffect()* hook szintaxisa.

a backend szerverről) a töltődést jelző komponens eltüntetését eredményezi. Ehhez mind a *response*, mind a *showLoader* változók részei kell legyenek a state-nek, *useState()* hívások által.

A felhasználó is létrehozhat saját hookokat, hogy speciális logikát tudjon újrahasznosítani a komponensek között. Ilyen a (4)-es kódrészletben látható *useDelete()* generikus függvény is. Itt először a generikus válasz és a hibát jelző változó state-hez rendelése történik. Ezt egy aszinkron függvény deklarációja követi, amely az adott API címre egy DELETE típusú REST kérést küld az *Axios* könyvtár segítségével [53], majd a kapott válasz alapján módosítja a két változót. A hook visszatéríti ezt a két változót és a hívást indító aszinkron függvényt. A hook a

```
const { response, showError, asyncDelete } = useDelete<ILabelResponse>();
```

szintaxissal hívható meg, az első két érték közvetlenül használható a komponens logikájában *if* utasításokhoz kötve és megjelenítve is, az *asyncDelete()* függvény pedig például a törlés gomb megnyomásához rendelt metódus részévé tehető. Hasonlóképp működik a GET kérésekért felelős hook is, azzal a két különbséggel, hogy még egy változót tartalmaz, amely a kérés kiküldése és a válasz érkezése között *true* értéket vesz fel; ez az adatok töltődését jelző komponenshez szükséges. Az aszinkron függvény itt nincs visszatérítve, hanem még egy, az URL-től függő *useEffect()* hook része, így az URL-ben található szűrők változtatása a keresés

```
const useDelete = <T extends {}>(): { response: T | undefined;
showError: boolean; asyncDelete: (url: string) => Promise<void> } => {
    const [response, setResponse] = useState<T | undefined>();
    const [showError, setShowError] = useState(false);
    const asyncDelete = async (url: string) => {
        const URL = process.env.REACT_APP_API_URL + url;
        await axios
            .delete<T, AxiosResponse<T>>(URL, { withCredentials: true })
            .then(
                (resp) => { setResponse(resp.data); },
                (error) => {
                    if (error.response) { setResponse(error.response.data); }
                    else { setShowError(true); }
                }
            );
    };

    return { response, showError, asyncDelete };
};
```

4. kódrészlet. A *useDelete()* saját hook.

komponenseiben valós időben frissíti a találatokat.

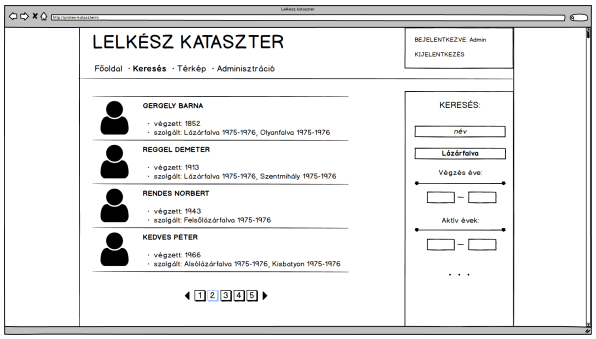
5.2. Reszponzív és átlátható UI

A projekt egyik fontos követelménye, hogy a weboldal kezelése minél intuitívabb és az adatok elhelyezése minél átláthatóbb legyen. Az ezzel foglalkozó *UX design* szakterület több módszerrel is igyekszik ezt az elsőre szubjektívnek is vélhető aspektust minél determinisztikusabbá tenni a felhasználói felület tervezése során.

A projekt kezdetén, a szakirodalom alapos kutatása után [39, 2], a csapat a *Balsamiq* nevű mockup-készítő és prezentáló szoftver [47] segítségével több különböző design tervet is készített az összes felület számára, amelyek közül a kliens és külső személyek véleménye alapján történt a választás. Két ilyen mockup látható az (5)-ös és a (6)-os ábrákon, amelyek a 7. fejezetben található képernyőképektől lényegében csak az oldalsáv elhelyezésében különböznek. Továbbá, a tényleges fejlesztés során is kaptunk visszajelzéseket az oldal használhatóságát illetően Jakab István UX szakértőtől.

Az szoftver platform nem rendelkezik natív Android vagy iOS klienssel. A csapat és a megrendelő közös véleménye szerint erre csak akkor lenne külön szükség, ha az okostelefonok valamilyen natív szolgáltatásaira építene a rendszer, mint például a kamera vagy a helymeghatározás. Ehelyett a felhasználói felület építésének lett fő szempontjává téve a reszponzív design, amely így bármilyen méretű képernyőn könnyen használható, legyen az nagyképernyő, notebook, táblagép vagy okostelefon. Néhány példa megtalálható a 7. fejezetben.

A reszponzív felület és az alapvető kinézet létrehozásában a *Semantic UI* könyvtár React-integrált verziója (továbbiakban SUIR) segített. Az weboldal felületének összes építőeleme ebből a könyvtárból származó újradefiniált HTML elemekből, vagy teljesen új komponensekből áll, legyen szó az űrlapok különböző beviteli mezőiről, tartalom elhelyezést befolyásoló komponensekről, kártyákról vagy gombokról. Ezek színe, mérete és egyéb



5. ábra. Design terv a lelkészek keresési oldaláról.

6. ábra. Design terv a lelkészek profiloldaláról szerkesztés módban.

vizuális tulajdonságai legtöbbször beépített változókon keresztül állítható be, de több esetben is manuálisan megadott CSS parancsokkal íródtak ezek felül, hogy jobban illeszkedjenek az oldalhoz. Ezek a komponensek mind *fluid-ok* ("folyékonyak"), tehát méretük az ablak méretéhez igazodik, miközben az összterület többnyire változatlan százalékát foglalják el.

A reszponzivitás érdekében tett lépések több különböző módszer segítségével valósulnak meg. A navigációs sávnak például kétféle implementációja van létrehozva, amelyek közül az egyik csak ikonokat tartalmaz. A két komponens között a SUIR *computer- / tablet- / mobile only* szelektorainak segítségével történik a választás. Táblagépen böngészve a különböző kártyák (a keresők és adatlapok esetében) egy oszlopba rendeződnek, a *Grid* szervezőelem *stackable* paramétere által. Az oldalak alapvető elrendezése okostelefonon is változik, a fő tartalom az egyébként bal oldalon lévő oldalsáv alá kerül, a kártyák pedig itt is egy oszlopban jelennek meg. Ehhez a felbontás figyelembe vétele a

```
@media only screen and (max-width: 768px) {...}
```

illetve hasonlóan a *min-width* CSS szabályok segítségével oldható meg.

A UX és UI dizájn között lényeges különbség, hogy előbbi nem csak a látható komponensekkel foglalkozik, hanem az oldal teljes használhatóságára és a funkcionalitások gördülékenységére is figyel. Jelen esetben ide sorolhatóak olyan apró kényelmi funkciók, mint a dupla-csúszkák alaphelyzetbe állításáért felelős gombok, a valós idejű keresési találatok, vagy a szűrők és az oldalszám a jelenlegi URL-be való illesztése a keresési oldalakon, hogy a pontos keresések és találatok a hiperlink által megoszthatóak legyenek. Utóbbi két funkció azon alapszik, hogy a JavaSriptben lehetséges az objektumok mezőin iterálni. Az (5)-ös kódrészletben megfigyelhető, ahogy a *filters* objektum funkcionálisan redukálódik egy, csak a nem alaphelyzetű szűrőket tartalmazó karakterlánccá, amely által új GET kérés küldődik a backend szerverre, illetve az utolsó sorban, a komponens megjelenítése előtt még, az

```
const [filters, setFilters] = useState(initFilters);
const filterURL = Object.entries(filters).reduce((prev: string, entry: [string, string]) => {
    if (
        entry[1] === "" ||
        (entry[0].includes("Start") && entry[1] === filterStartYear.toString()) ||
        (entry[0].includes("End") && entry[1] === filterEndYear.toString())
    ) {
        return prev;
    } else {
        return prev + entry[0] + "=" + entry[1] + "&";
    }
}, "");
const { response, loading, showError } = useFetchData<IPriestMin[]>(`/priests?${filterURL}`);
return (<>
            <Redirect to={`/priests/search?${filterURL}pageNr=${pageNr}`} />
            // ...
```

5. kódrészlet. A szűrés mezőinek használata a kényelmi funkciók megteremtéséért.

oldal navigál a változtatott útvonalra. Ez több más technológia esetében végtelen újratöltési ciklust eredményezne, a React viszont csak a state változására reagál az érintett komponensek újratöltésével, ezért csak az URL változik.

A weboldal pontos színeinek és betűtípusának finomhangolása érdekében a SUIR egy *craco* nevű könyvtár segítségével újra lett kompilálva, hogy elérhetőek legyenek a belső konfigurációs állományai is [24].

5.3. Autentikáció és autorizáció

A backend szerver előző fejezetben tárgyalt biztonsági megoldásai mellett a web kliens is több módszert alkalmaz az adatok védelmére a be nem jelentkezett vagy nem megfelelő hozzáférési szinttel rendelkező felhasználókkal szemben.

Ezek közül a leginkább szembetűnő a navigációs sáv változása a bejelentkezés és a fiók típusának függvényében. Hasonlóképpen vannak elrejtve az adatmanipulációval kapcsolatos vezérlők is az elérhető oldalakon belül, ilyen az adatlapokon megjelenő *Szerkesztés* gomb, vagy a főoldalon a hírek hozzáadásának, szerkesztésének és törlésének ikonjai. Így a nem autorizált funkcionalitások meg sem jelennek az oldalakon.

Ennél egy lépéssel tovább megy a *react-router* könyvtár *Route* komponense köré írt saját *GuardedRoute* függvény. Az applikáció gyökerében található *Switch* (elosztó) komponensben felsorolt utak ezzel lettek beburkolva [17]. A (6)-os kódrészletben lévő komponens megkapja a *Route* számára szükséges *path* és *render* értékeit, ezen kívül viszont átadódik a bejelentkezett felhasználó adatait tartalmazó *user* objektum és a szóban forgó oldalt elérni engedélyezett felhasználói típusok listája is. Utóbbi kettő segítségével el tudja dönteni, hogy a *Route* komponenst visszatérítheti-e úgy, hogy a kért útvonalra mutasson, vagy hogy ez az oldal a felhasználó számára egy hibaüzenetet tartalmazó komponenst töltsön be. Ezzel is védve vannak az amúgy fizikailag is elrejtett navigációs opciók, hogy az URL birtokában se lehessen elérni őket.

Az autentikációt érintően, a bejelentkezett felhasználó szesszióját kényelmi okokból

```
<GuardedRoute
    path="/priests/profile/:id"
    render={({ match }) =>
        <PriestProfilePage
            priestID={match.params.id}
            userRole={user ? use.role : undefined} />
    }
    user={user}
    roles={[UserRole.READER, UserRole.EDITOR, UserRole.ADMIN, UserRole.OWNER]}
/>
```

6. kódrészlet. A *GuardedRoute* saját komponens használata.

kijelentkezésig megőrzi a böngésző. Ennek a be- illetve kijelentkezéskor történő kezelése egy saját hook keretein belül ágyazódik a weboldal megfelelő részeibe. Ez a függvény felelős a szesszió megszerzéséért, tárolásáért és törléséért, valamint a *user* objektum state-hez rendeléséért is.

A regisztráció és a bejelentkezés felületein az űrlapok dinamikus hibaüzenetei egy fordítási kulcsokat tartalmazó lista alapján jönnek létre. Ez a lista az űrlapok beküldését kezelő függvényben épül fel a különböző hibaforrások összesítése által: hiányzó mezők, helyileg hibásnak vélt adatok (e-mail cím formája, jelszavak megegyezése) és a backend szerverről visszaérkezett hibaüzenetek.

5.4. Dátumok kezelése és megjelenítése

A dátumok helyes kezelése a kereső funkcionalitások teljes működése miatt kulcsfontosságú feladat volt. Az eredeti adatbázisban karakterláncként voltak lementve, és ebből adódóan rengeteg elírást ("19885.04.28"), hiányosságot ("1992 január eleje") és nem ide tartozó információt ("1919.10 - segédlelkész") tartalmaztak.

A tisztítás érdekében egy új struktúra lett létrehozva, amely külön mezőkben tárolja az évet, a hónapot, a napot és a régi, karakterlánc értékét is. Az adatok átalakítása egy reguláris kifejezésekre épülő algoritmus által történik, a backenden történő importáláskor. Első lépésben mindhárom számérték felismerésével próbálkozik a rendszer, ha ez nem sikerül, akkor az év és hónap, és végül csak az évszám kinyerésére tesz kísérletet. Ez a módszer eredményesnek bizonyult, az évszámok 95%-át sikerült felismerni.

Az ilyen módon tisztított és normalizált dátumok megjelenítése a webes kliens felületén két komponens és két másik segédfüggvény segítségével valósul meg. Ezek lényege, hogy a dátumok kezelését végző kód könnyen újrahasznosítható legyen.

A *DateInput* a négy adattag (év, hónap, nap, karakterlánc) számára négy beviteli mezőt, illetve egy kondicionálisan megjelenő hibaüzenetet tartalmaz. Az utóbbi akkor jelenik meg, ha a felhasználó helytelenül beírt (hibás hónap és nap kombináció, szökőévek stb.) dátumot próbál elmenteni a szerkesztés módban. A hibaüzenet a dátum mellett és a szerkesztést elmentő vagy visszavonó gombok alatt is megjelenik, ezen felül maga a szerkesztés mód sem zárható be mentéssel ha valamelyik dátum még hibás.

A *DateDisplay* komponens próbálja a dátumokat minél pontosabban megjeleníteni. Ha mindhárom számérték elérhető akkor teljes dátumot térít vissza, hiányzó nap, vagy nap és hónap esetén pedig csak az évet és a hónapot, illetve csak az évet. Az egér föléhúzásakor az utóbbi két esetben a kezdeti adatbázisból származó karakterlánc jelenik meg egy szövegbuborékba ágyazva. Amennyiben még az évszámot sem sikerült kinyerni, az eredeti karakterlánc jelenik

meg a dátum helyett, és ha az is üres volt akkor pedig egy *Nincs adat* felirat.

5.5. Nemzetköziesítés

Az oldalon található szövegek a *react-intl* könyvtár használata által jelenhetnek meg több (jelen esetben két) nyelven [9]. Az angol és magyar nyelvű szövegek két JSON fájlban található kulcs-érték párosok, ezek elérését az alkalmazás egyik legkülső rétegeként instanciált *IntlProvider* komponens biztosítja minden alkomponens számára. A felső navigációs sávon található lenyíló menüben kiválasztott nyelv értéke egy saját hook segítségével ágyazódik be az *IntlProvider* szerkezetébe. Ugyancsak ez a hook a böngésző lokális tárolójába menti a legutolsó választást és az oldal újbóli felkeresésekor először innen próbálja elérni azt, ezáltal a látogatások között megmarad a felhasználó által preferált nyelv.

A közel 350 különböző kulcshoz rendelt megfelelő nyelvű szövegeket az oldal elemei a *FormattedMessage* komponens illetve funkcionálisan a *useInt()* hook által visszatérített függvény segítségével érik el. Az utóbbi karakterláncot térít vissza, így használható olyan kontextusban is, ahol egy komponens számára kell paraméterként megadni szöveget.

Lehetséges változók beágyazása a fordításokba, a

```
<FormattedMessage id="search.parishNoCoordinates" values={{ nrMap }} />
```

komponens például a kulcshoz rendelt

```
", ebből {noMap, number} koordináták nélkül"
```

fordítás megfelelő részére illeszti a találatok közül a koordinátákkal rendelkező gyülekezetek számát.

Megemlítendő még, hogy a React alapelveinek megfelelően ezek a komponensek is felülírhatóak. A projektben három ezekre ráépülő komponens is van, amelyek a lefordított szöveget dőltté, aláhúzottá vagy vastaggá teszik és a paraméterek alapján további formázásokat is alkalmazhatnak. Ilyen minimális módosítást alkalmazó komponensek segítségével marad a komplexebb fájlokban lévő kód is átlátható.

5.6. Az ábrakészítőről

A *Reaviz* könyvtár [21] a weboldal adatvizualizáció-készítő felületén van felhasználva. A projekt szempontjából legfontosabb előnye, hogy az összes ábratípus esetében ugyanazt az adatmodellt használja, ezért egyszerűen és tömören történhet az adatok aggregálása bármelyik ábra számára.

Ez az aggregáció funkcionálisan van implementálva, iteráló (*forEach*), redukáló (*reduce*), szűrő (*filter*) és rendező (*sort*) függvények kombinálásának eredményeként, a felhasználó által

```
priests.forEach((p: IPriestChartData) => {
    if (p[xAxis] !== undefined && p[xAxis] !== 0 && p[xAxis] !== "") {
        usedDataSize += 1;
        const match = chartData.filter((item: IChartData) => item.key === p[xAxis]);
        if (match.length === 0) {
            chartDatum = { key: p[xAxis], data: p[yAxis], counter: 1 };
            chartData.push(chartDatum);
        } else {
            match[0].data += p[yAxis];
            match[0].counter += 1;
        }
    }
});
// ...
const temp = chartData.filter((item: IChartData) => item.counter < minGroupSize);
chartDatum = {
    key: fm({ id: "charts.etc" }),
    data: temp.reduce((prev: number, next: IChartData) => prev + (next.data || 0), 0),
    counter: temp.reduce((prev: number, next: IChartData) => prev + (next.counter || 0), 0)
};
```

7. kódrészlet. Az ábrákhoz szükséges adatok feldolgozásának első két lépése.

kiválasztott és dinamikusan elért adatmezőkön. Ennek első két és leglényegesebb lépése a (7)-es kódrészletben figyelhető meg. A visszatérített adatokon történő iteráció során a *Kategória tengelyként* kiválasztott *xAxis* és a *Csoportosítás típusaként* beállított *yAxis* karakterláncok segítségével érhetőek el az entitás célzott adattagjai. A filter() hívással van ellenőrizve, hogy a jelenlegi kategória már létezik-e a már aggregált adatok között, és ennek függvényében inkrementálódik a talált érték, vagy jön létre egy új csoport.

Ugyancsak a (7)-es ábrán, a kihagyott rész alatt a kis csoportok összesítése következik, amely az ábra tisztítását szolgálja. Ennek küszöbe és az ábrán való megjelenítése a felhasználó által konfigurálható. Az első sor kiválogatja a beállított értéknél kisebb csoportokat. Ennek alapján egy új, összesített kategória jön létre a hátralevő sorokban, amely *reduce()* hívások segítségével számolja össze az ehhez tartozó értékeket.

Az algoritmus hátralévő részében többek közt az adat és a számláló alapján ki vannak számolva a megjelenítendő végső értékek, a tisztítás küszöbe szerint kiszűrődnek a kisebb csoportok, és név vagy érték alapú rendezés hajtódik végre.

A könyvtár másik nagy előnye, hogy *opinionated*, azaz "csökönyös": az ábrák megjelenéséről és témázásáról nagyrészt maga dönt, így ennek adatspecifikus implementációja nem hárul a fejlesztőre.

6. Fejlesztési eszközök és módszerek

A fejezet a projektben felhasznált fejlesztési módszereket és eszközöket mutatja be.

6.1. Agilis szoftverfejlesztés

A projekt kezdetén a csapat egy elterjedt agilis módszer mellett döntött. Az inkrementális és iteratív fejlesztés alapelvei miatt esett a *Scrumra* [22] a választás, ahol a fejlesztés menete néhány hetes, *sprintnek* nevezett intervallumokra osztható.

Minden egyes ilyen szakasz kezdetén, a *sprint planning* keretein belül, a csapattagok meghatározzák a teendőket, majd bonyolultság és relevancia szerinti priorizálás után a *sprint backlogba* kerül annyi, amennyit a csapat a sprint végéig előreláthatóan be tud fejezni. A sprint során minden nap rövid megbeszélésre kerül sor, ami segíti a fejlesztőcsapatot, hogy naprakészek legyenek egymás munkájával vagy problémákat oldjanak meg közösen. A sprint végén a funkcionalitások bemutatására kerül sor a megrendelő jelenlétében, majd ezután egy retrospektív gyülés jön, melyen a csapat átbeszélheti az elmúlt hetek pozitív és negatív tapasztalatait, illetve változtatásokat javasolhatnak a következő sprintre a jobb csapatmunka és előrehaladás érdekében.

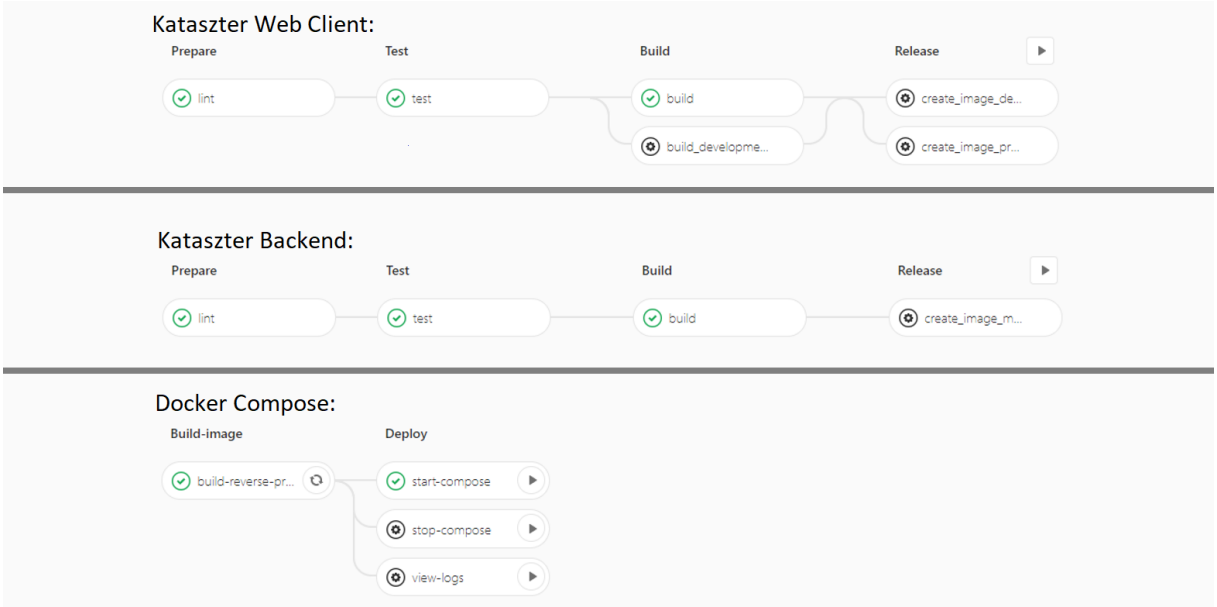
A fejlesztés során a feladatok állapotának nyilvántartása egy *kanban board*-on történt. A *kanban tábla* által a csapat vizuálisan is követheti a különböző feladatok haladását [42]. Ezek mint kártyák jelennek meg a táblán és oszlopról oszlopra haladnak. A *sprint planning* során meghatározott feladatok a legelső, *sprint backlog* oszlopba kerülnek. A csapattagok kiválasztják és magukhoz rendelik a számukra legmegfelelőbb feladatot és ezt a *doing* oszlopba helyezik. Miután a feladattal végeztek a *review* oszlopba helyezik át. Ellenőrzés után a feladat a *merged* oszlopba kerül, majd a sprint végeztével, minden olyan kártya amely a *merged* oszlopban van és a megrendelő által elfogadottnak minősül, átkerül a *closed* oszlopba.

6.2. Verziókövetés, folyamatos integráció és kitelepítés

A *Continuous Integration* [43] vagy *folyamatos integráció* nagy szerepet kapott a projekt fejlesztése során. Ez engedélyezte, hogy automatikus kompilálások, tesztek, kódellenőrzések fussanak mindegy egyes változtatás után, azonnal jelezve, ha bármi hiba csúszott a rendszerbe, vagy ha nem voltak betartva a kódolási konvenciók.

A Gitlab-ba [49] beépített CI/CD[13] eszköz [11] segítségével a (7)-es ábrán láthatóakhoz hasonló *pipeline*-ok hozhatók létre, amelyeken belül különböző munkafolyamatok szabhatóak meg. A projekt esetében, amikor egy fejlesztési ágra új kód kerül fel, elindul egy pipeline,

[13]Continuous integration and Continuous Deployment



7. ábra. A három CI/CD pipeline.

amely egymástól függő feladatokból, úgynevezett *jobokból* áll. Elsőnek a statikus kódellenőrzés fut le, backenden a *golang-ci lint* [32], frontenden pedig a *TSLint* [45] által, amely ellenőrzi, hogy a kódolási konvenciók be lettek-e tartva. Ezután egy új job hívódik meg, mely lefuttatja a teszteket. Ezt követően a *build* jobon belül kompilálódik az alkalmazás. Ezen kívül még egy kézzel indítható *release* job is meghívható. Ez a kitelepítés automatizálásában játszik szerepet. Mikor a release folyamat elindul, létrejön egy új *docker image* [4]. Ez a rendszerkép feltöltődik a Gitlab saját regiszterébe, amelyen keresztül a harmadik pipeline el tudja érni mind a backend, mind a frontend elkészült image-eit.

A teszt folyamaton belül a backenden Golang-ben írt tesztek futnak le a repository réteg komponensein. A lefedettség itt 100%-os, amit egy shell script számol össze a Golang által csomagonként kiállított lefedettségi értékek alapján. Ez minden fejlesztési ágra való feltöltéskor lefut.

A backend szerver részletes tesztelése egy *Postman* nevű szoftver [50] segítségével történik, ami lényegében API kéréseket szimulál és előre konfigurált részleteket tesztel a kapott válaszokon, mint válaszidő, státuszkód vagy magára a kapott adatra vonatkozó ellenőrzések. Itt jelenleg 87 kéréshez társul összesen 260 teszteset, mely magába foglalja az összes elérhetővé tett API végpontot és ezeknek a különböző körülmények közt megkísérelt elérését, beleértve be nem jelentkezett vagy nem megfelelő felhasználói szinttől érkező kéréseket is. Ezek az API tesztek jelenleg csak lokális környezetben futtathatóak, automatizálásuk továbbfejlesztési lehetőség.

A frontend részen *snapshot testing* [46] történik, ami biztosítja azt, hogy a felhasználói felületen nem ment végbe váratlan változás. Egy tipikus ilyen teszt elkészít egy bekonfigurált UI komponenst, melyet lement és a későbbiekben ehhez hasonlítja az azonos paraméterek

által újra elkészített komponenst [46]. Amennyiben a *snapshotok* frissítése nélkül történt valamilyen változás, buknak a tesztek. A tesztesetek során a komponens izolálva van, tehát minden olyan komponens, amely használva van az aktuálisan tesztelt komponensben, *mockolva* lesz, azaz a használt komponens vagy függvény egy, csak a viselkedést utánzó objektum által helyettesítődik. A backendhez hasonlóan itt is megtörténik a lefedettség mérése. A backend és frontend CI pipeline-jai a projekt eddigi fejlesztése során több mint 1100 alkalommal futottak le.

A *Continuous Deployment* [43] vagy folyamatos kitelepítés a backend és frontend CI pipeline-jainak utolsó feladatai által valósul meg, egy harmadik külön pipeline-ban. Ez a folyamat az elkészült és a Gitlab saját regiszterébe feltöltött rendszerképekből felépített docker konténereket összeköti egymással, majd kitelepíti az alkalmazást a tesztszerverre. Ez a kitelepítés a *Docker Compose* eszköz [8] segítségével valósul meg, mely a fent említett konténerek számára egy belső hálózatot hoz létre, hogy a folyamatok kommunikálhassanak egymással. A kitelepített alkalmazás a külvilág számára egyetlen közös portot nyit meg egy *proxy szerveren* keresztül, melynek feladata a közös címre beérkező kérések elosztása a backend szerver és a web kliens között.

6.3. Segédeszközök

A **Visual Studio Code** egy letisztult és könnyen konfigurálható környezet, a frontend és backend implementálásához is erre esett a választás. A rengeteg elérhető kiegészítő mellett biztosít szintaxis kiemelést, kódkiegészítést, beépített parancssort illetve git és docker integrációt is [52].

A **GitKraken** a git verziókövető rendszerhez biztosít egy vizuális kezelőfelületet [48], ezáltal könnyebbé téve a git parancsok végrehajtását és a különböző verziók, ágak vagy repositoryk közötti váltást.

A nemzetköziesítés szempontjából az **i18n-editor** nevű applikáció [38] könnyítette meg a munkát. Egy egyszerűen kezelhető vizuális felület segítségével engedélyezte a nemzetköziesítéshez szükséges különböző nyelvi fájlok azonos idejű módosítását, így téve átláthatóbbá a fordítások létrehozását és szerkesztését.

7. A SoulMind működése

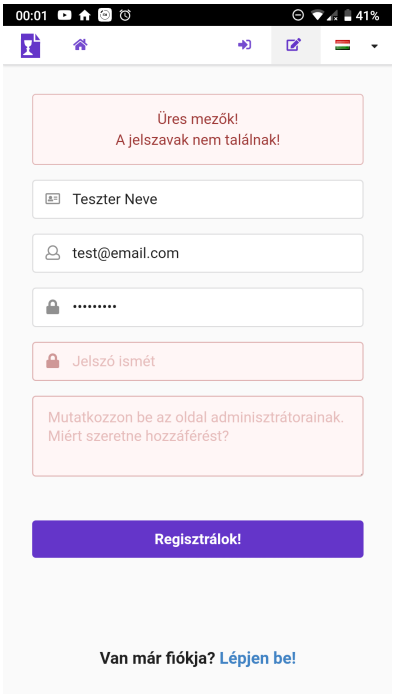
Az állandó navigációs sáv a bejelentkezés előtt csak a belépés, a regisztráció és a főoldal linkjét tartalmazza a nyelvválasztón kívül. Bejelentkezést követően, a felhasználói szint függvényében további új funkcionalitások érhetők el.

A be nem jelentkezett felhasználók számára a navigációs sáv bal oldalán érhető el a (8)-as ábrán is látható regisztrációs oldal. Bejelentkezéskor a felhasználónak lehetősége van új jelszót igényelni az *Elfelejtett jelszó* feliratra kattintva.

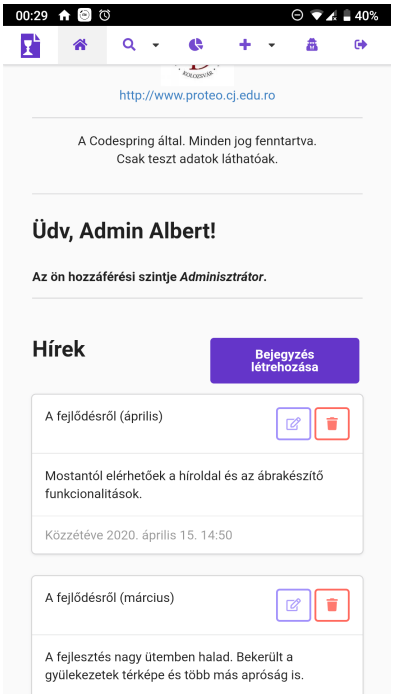
Az 1. fejezetben tárgyalva voltak a különböző felhasználói szerepkörök és a hozzájuk kapcsolt különböző funkcionalitás-csoportok. A dolgozatban az adminisztrátor szempontjából lesz bemutatva a működés, hogy az összes fontosabb funkcionalitás helyet kapjon a leírásban, de néhány képernyőfotón az alacsonyabb szintű jogosultságokkal bíró felhasználók számára megjelenített menü is megfigyelhető. Bejelentkezés után a (9)-es ábra egy teljes összképet ad az elérhető oldalakról.

Mint adminisztrátor, a főoldalon elérhetővé válik a *Bejegyzés létrehozása* gomb, illetve a szerkesztést és a törlést indító ikonok. Előbbi kettőre kattintva a (10)-es ábrán látható felület nyílik meg, ahol megadható a bejegyzés címe, és tartalma, az időbélyeg viszont automatikusan illesztődik majd hozzá.

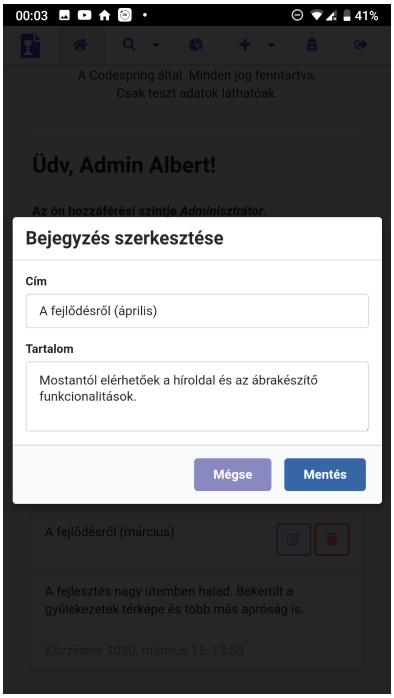
A menüpontok között a *Keresés* lenyíló menüben választhatunk lelkészek vagy gyülekezetek szűrése közül. A lelkészek keresési felülete a a (11)-es ábrán látható. Itt a lelkészek egy sornyi



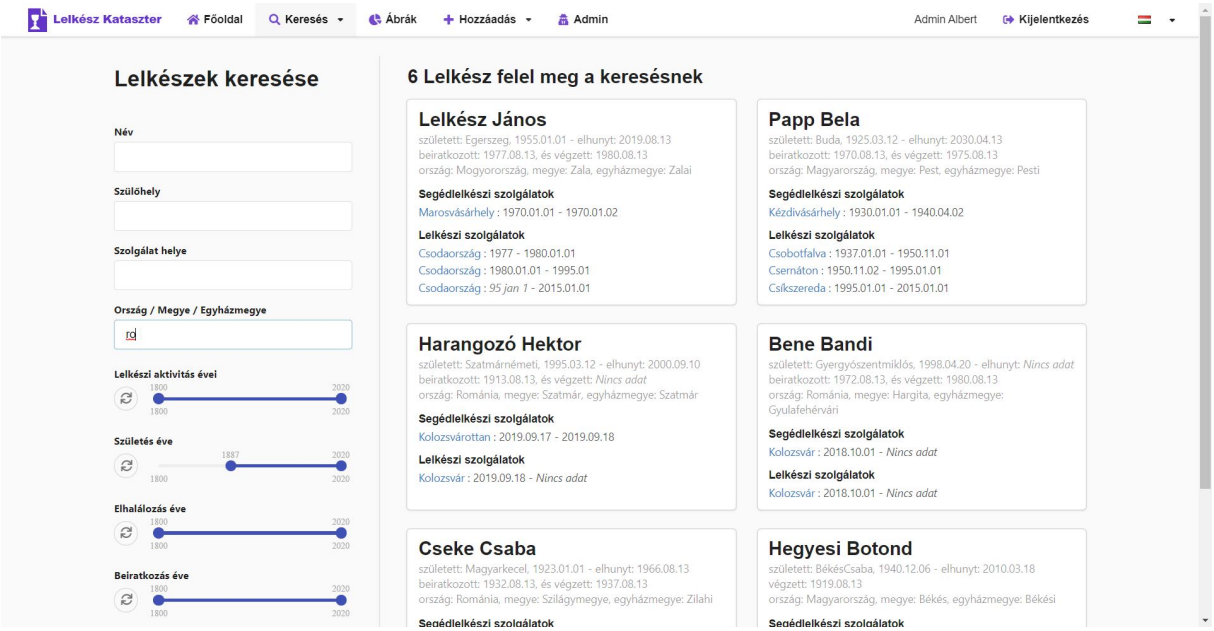
8. ábra. Regisztrációs oldal, hibákkal.



9. ábra. Az alkalmazás főoldala a hírekkel.



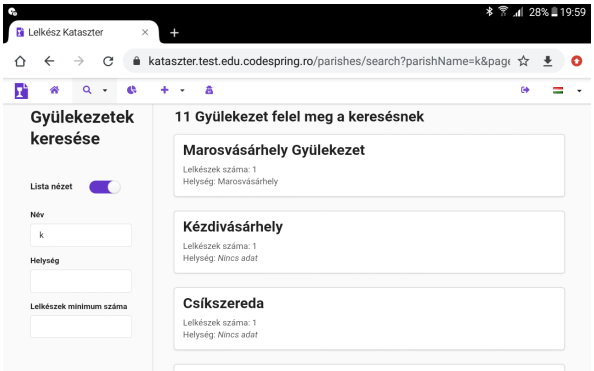
10. ábra. Hír szerkesztése egy adminisztrátor által.



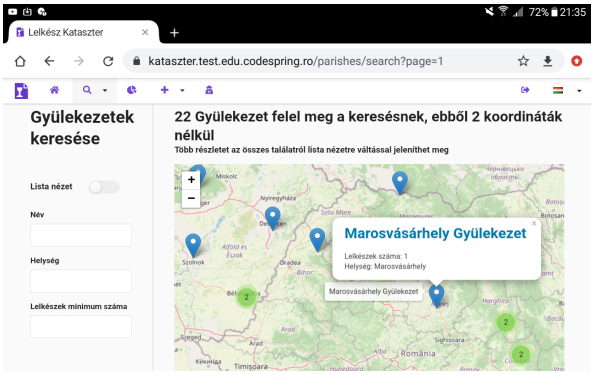
11. ábra. Lelkészek keresésének oldala a szűrési lehetőségekkel és az eredményekkel.

különböző tulajdonság alapján szűrhetőek és célzottan kereshetőek a bal oldali komponensek által. Ezek között a szűrők között találhatóak szöveges beviteli mezők, ahol konkrét adatok alapján történhet a szűrés. Egyes beviteli mezők több adattagot is figyelembe vesznek. A dupla-csúszkák segítségével állíthatóak be időintervallumok. A keresés eredményei a jobb oldalon jelennek meg, valós időben reagálva a szűrők bármilyen változtatására. A találatok listája tízesével oldalakra van osztva. A kártyák alatt lévő vezérlőkkel lehet az oldalak között navigálni. Az alapadatokat tartalmazó kártyákra kattintva a felhasználó a lelkész adatlapjára irányítódik át (amennyiben a fiókja rendelkezik megfelelő jogosultsággal). Ugyanezeken a kártyákon a gyülekezetek nevei is kattinthatóak, ezek a hivatkozások a hozzájuk tartozó gyülekezet profilhoz vezetnek.

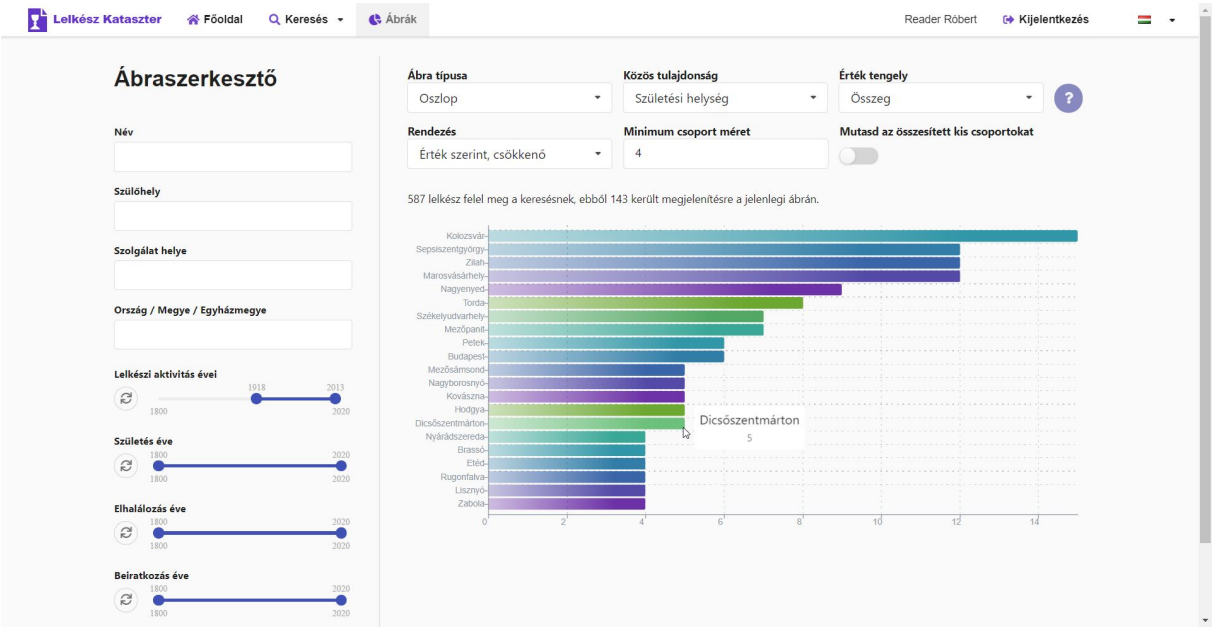
Ha a keresés a gyülekezetek között történik, hasonlóképpen mint a lelkészeknél, különböző szűrők által valósul meg. Az eredmények megjelenítése kétféleképpen történhet.



12. ábra. Gyülekezetek keresése lista nézetben táblagépen.



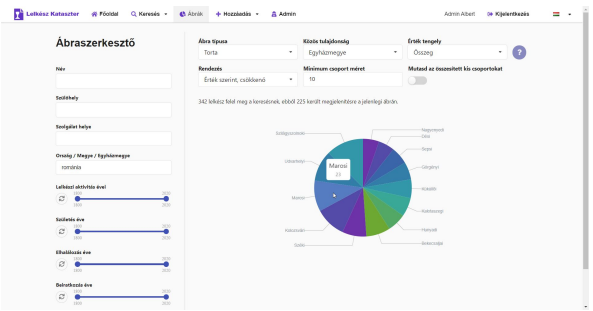
13. ábra. Gyülekezetek keresése a térképen.



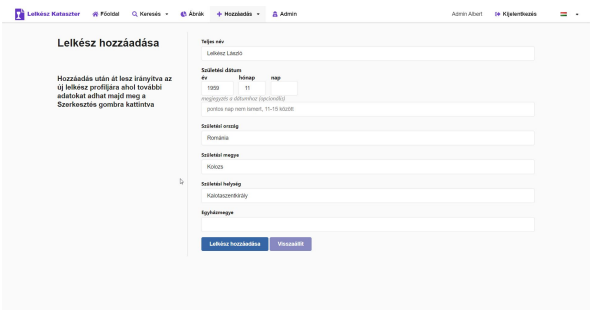
14. ábra. Oszlopdiagram az 1918-2013 között aktív lelkészek születési helyéről, csak a 4-nél nagyobb csoportok.

A lelkészekéhez hasonló megjelenítési módszer a (12)-es ábrán is látható. Lista nézet helyett egy térképen is megjeleníthetőek az adatok. Mivel egymáshoz közel vagy akár egy helységen belül több gyülekezet is létezhet, ennek megoldására a marker helyett egy szám jelenik meg, ez a lefedett régió gyülekezeteinek száma. Erre ráközelítve, a szám több marker-re bomlik szét amint a (13)-as ábra is mutatja. A listában lévő vagy a marker-eken felugró kártyák kattinthatóak, ezek a hivatkozások a hozzájuk tartozó gyülekezet profilhoz vezetnek.

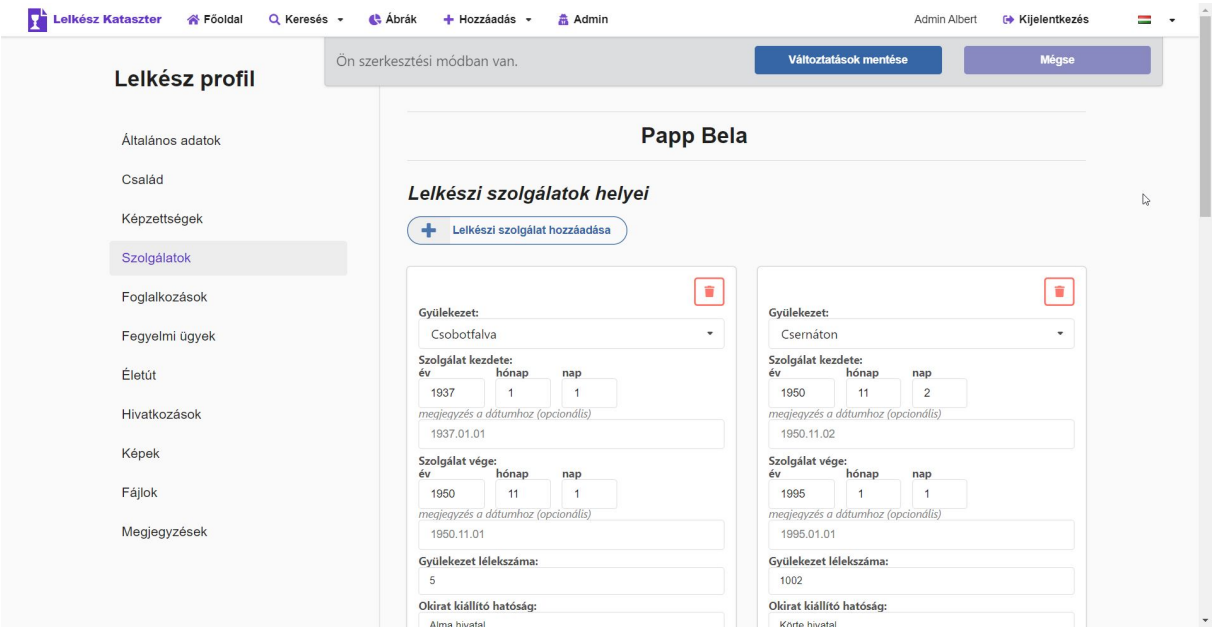
A menüpontok közül a következő választható elem az ábrakészítőre visz. A (14)-es képen is megfigyelhető oldalon egyszerűen lehet különböző típusú adatvizualizációt készíteni. Az itt elérhető opciók segítségével részletesen testre szabhatóak a vizualizáció paraméterei. Ezek közül a legfontosabb az a két lenyitható menü, ahol az ábrán megjelenítendő adatmezők választhatóak ki. Itt az első opció azt rögzíti, hogy melyik mező alapján kell csoportosítani az adatokat, ez többek között lehet például *születési év* vagy *helység*. A második lenyíló



15. ábra. Tortadiagram a romániai lelkészek egyházmegyéiről az ábrakészítő oldalon, csak a 10-nél nagyobb csoportok.



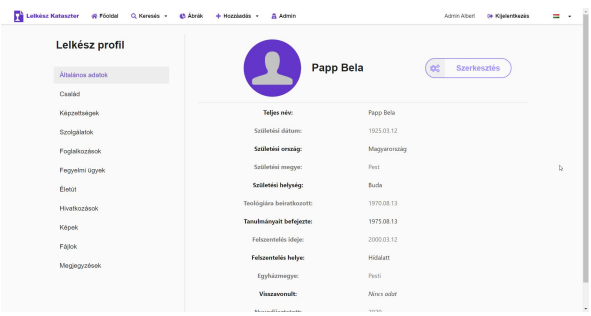
16. ábra. Új lelkész adatainak bevezetése.



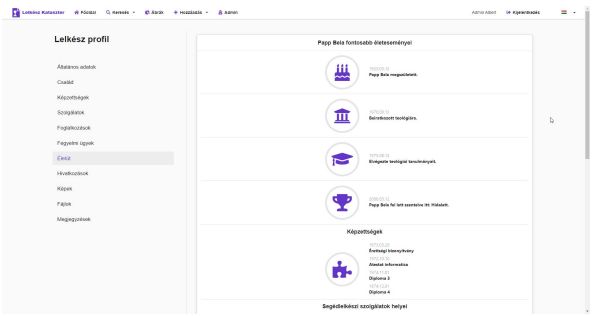
17. ábra. A lelkész szolgálatainak szerkesztése.

menüben pedig a kategóriákhoz rendelt értékről kell dönteni, ami lehet egyszerűen a kategória *mérete*, de különböző átlagolt értékeket is felvehet, mint *gyerekek* vagy *szolgálatok száma*. Továbbá kiválasztható az ábra típusa: torta-, oszlop- vagy vonaldiagram, előbbi két esetében pedig három másik beállítás érhető még el, az adatok rendezését és tisztítását (kis méretű csoportok összesítése vagy kihagyása) illetően. A különböző opciók kiválasztásának kombinálhatósága miatt jelenleg 630 féle különböző ábra hozható létre egy adathalmaz alapján. Egy tortadiagramra a (15)-ös ábrán látható példa. Ezenkívűl, az itt felhasználni kívánt adathalmaz szűkíthető is a nemrég tárgyalt keresés komponens által, mely itt is jelen van.

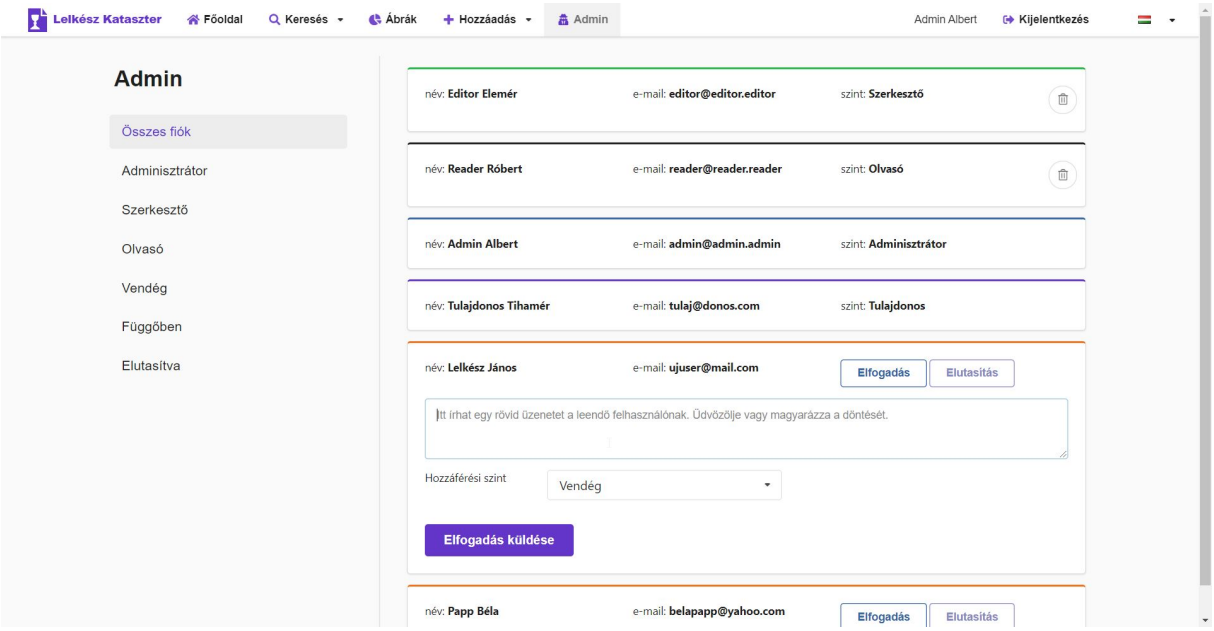
Folytatva a menüpontok sorozatát, a következő választható elem a *Hozzáadás*. Itt ismét lelkész vagy gyülekezet létrehozása közül kell választani. Amennyiben új lelkész hozzáadására kerül sor, a (16)-os ábrán látható oldal fogadja a felhasználót. Ezen az oldalon a lelkész fontosabb adatait lehet megadni, és ezáltal létrehozni a lelkész entitást. A létrehozást követően, az oldal átirányít a lelkész profiljára, ahol a további mezők kitöltése is lehetővé válik. Hasonlóan történik az új gyülekezet hozzáadása is.



18. ábra. A lelkész profil alapadatai.



19. ábra. A lelkész életútja.



20. ábra. Adminisztrátori felület.

Szó esett már többször is a lelkészek és gyülekezetek profiljáról. Ezekre több hivatkozás is mutat a rendszeren belül. Lelkészek esetében, egy ilyen hivatkozásra kattintva a felhasználó elé a (18)-as ábrán látható oldal kerül. Mindkét adatlapon az oldalsávban különböző kategóriák érhetők el, melyek alapján az adatok fel lettek osztva. A lelkész szolgálataira kattintva láthatóvá válik, ahogy ezt a (17)-es ábra is mutatja, hogy az információk ezen belül is csoportosulnak. A lelkész minden egyes szolgálati gyülekezetéhez további információk tartoznak, amelyek mind külön kártyákra kerülnek. Ez a család, képzettségek, foglalkozások, fegyelmi ügyek és hivatkozások oldalain is hasonlóképpen épül fel. Amennyiben a felhasználó jogosult szerkesztésre, az előző ábrán látható felületen módosíthatja a lelkész profilját. Kiemelendő itt a lelkész életútját bemutató menüpont, mely a meglévő adatokból generál egy időrendbe rendezett áttekintést a lelkész főbb életeseményeiről, ez a (19)-es ábrán követhető. A gyülekezetek profil oldala hasonló, itt csak három aloldal van, az általános adatokról, képekről és letölthető fájlokról.

Megnyitva az admin oldalt a (20)-as ábra tartalma tárul az adminisztrátor elé. A kártyákon különböző felhasználók adatai láthatóak. Amennyiben egy felhasználó beküldte a regisztrációját, minden egyes adminisztrátor kap erről egy értesítő e-mailt, majd itt dönthet az adott felhasználó kérésének elfogadásáról. Elutasíthatja, vagy amennyiben elfogadja a kérelmet, egy szerepkört is rendel a felhasználó mellé. A szerepkörökhöz tartozó színeken kívül az oldalsáv is segít a gyors eligazodásban.

Következtetések és továbbfejlesztési lehetőségek

A projekt főbb célkitűzései mind megvalósultak: a létrehozott platform nyújtja az elvárt funkcionalitásokat a lelkészek és gyülekezetek nyilvántartását és a kereshetőséget illetően, könnyen átlátható és használható. A regisztráció-elfogadás alapú, többféle szerepkört megkülönböztető felhasználói rendszer részletes kontrollt tesz lehetővé az adatok fölött.

Közel egy év fejlesztés után is van még olyan tervezett továbbfejlesztés, amelynek ötlete már a projekt kezdeti szakaszában felvetődött. Ugyanakkor, vannak menet közben felmerült ötletek, amelyek már részei is a platformnak. A fejlesztői csapat véleménye szerint ez az agilis fejlesztés helyes gyakorlatba ültetésének eredménye.

Ahogy a terv sem lehet végleges, úgy az agilis módszertan alapelvein alapuló szoftverfejlesztés esetében sem igazán lehet teljesen kész termékről beszélni.

A dolgozat írásának idejében a soron következő fejlesztések és javítások listája magába foglalja az egy lelkész életét végigkövető térkép létrehozását, illetve a lelkész profiloldal PDF-ként való mentésének lehetőségét.

Egy másik nagyobb tervezett funkcionalitás a szerkesztési javaslatok bevezetése lenne. Ennek keretén belül a legalább *Olvasó* rangú felhasználók bizonyos adatmezőket bizonytalannak jelölhetnének, vagy javaslatokat tehetnének hibás vagy hiányzó adatok értékeire, a mező új értékével és egy indoklással. A javaslatokat a legalább *Szerkesztő* szerepkörű felhasználók elbírálhatják. Ez segítene a weboldalra regisztrált lelkészeknek abban, hogy saját vagy ismerőseik profiljának teljességéhez hozzá tudjanak járulni.

Be van tervezve még pár apróbb technikai javítás is, mint például az API tesztek a *folyamatos integrációba* illesztése, vagy a hiba-propagáció optimalizálása.

Hivatkozások

[1] G. Acharya. *How to Send Email through Gmail SMTP Server Using GO*. URL: https://pepipost.com/tutorials/send-email-through-gmail-smtp-server-using-go/ (elérés dátuma: 2020. jan. 2.)

[2] N. Babich. *Most Common UX Design Methods and Techniques*. 2017. júl. 13. URL: https://uxplanet.org/most-common-ux-design-methods-and-techniques-c9a9fdc25a1e (elérés dátuma: 2020. ápr. 30.)

[3] K. Ball. *The increasing nature of frontend complexity*. 2019. jan. 30. URL: https://blog.logrocket.com/the-increasing-nature-of-frontend-complexity-b73c784c09ae/ (elérés dátuma: 2020. ápr. 30.)

[4] E. Boersma. *Docker Image vs Container: Everything You Need to Know*. 2019. máj. 3. URL: https://stackify.com/docker-image-vs-container-everything-you-need-to-know/ (elérés dátuma: 2020. máj. 1.)

[5] P. Le Cam. *Introduction - React-Leaflet*. URL: https://react-leaflet.js.org/docs/en/intro (elérés dátuma: 2020. ápr. 30.)

[6] M. Chan. *SQL vs. NoSQL – what's the best option for your database needs?* URL: https://www.thorntech.com/2019/03/sql-vs-nosql/ (elérés dátuma: 2020. ápr. 30.)

[7] F. Copes. *What is a Single Page Application?* 2018. nov. 11. URL: https://flaviocopes.com/single-page-application/ (elérés dátuma: 2020. ápr. 30.)

[8] Docker hivatalos dokumentáció. *Overview of Docker Compose*. URL: https://docs.docker.com/compose/ (elérés dátuma: 2020. máj. 1.)

[9] Format.JS hivatalos dokumentáció. *react-intl: Overview*. URL: https://formatjs.io/docs/react-intl (elérés dátuma: 2020. ápr. 30.)

[10] Gin Web Framework hivatalos dokumentáció. *Introduction*. URL: https://gin-gonic.com/docs/introduction/ (elérés dátuma: 2020. ápr. 30.)

[11] GitLab hivatalos dokumentáció. *GitLab CI/CD*. URL: https://docs.gitlab.com/ee/ci/ (elérés dátuma: 2020. máj. 1.)

[12] Golang hivatalos dokumentáció. *The Go Programming Language*. URL: https://golang.org/doc/ (elérés dátuma: 2020. ápr. 30.)

[13] Google Maps Platform hivatalos dokumentáció. *Place Search*. URL: https://developers.google.com/places/web-service/search (elérés dátuma: 2020. ápr. 30.)

[14] MongoDB hivatalos dokumentáció. *Structure your Data for MongoDB*. URL: https://docs.mongodb.com/guides/server/introduction/ (elérés dátuma: 2020. ápr. 30.)

[15] MongoDB hivatalos dokumentáció. *The MongoDB 4.2 Manual*. URL: https://docs.mongodb.com/manual/ (elérés dátuma: 2020. ápr. 30.)

[16] Oracle hivatalos dokumentáció. *What Are RESTful Web Services?* URL: https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html (elérés dátuma: 2020. ápr. 30.)

[17] React Training hivatalos dokumentáció. *React Router: Declarative Routing for React.js*. URL: https://reacttraining.com/react-router/web/guides/primary-components (elérés dátuma: 2020. ápr. 30.)

[18] React hivatalos dokumentáció. *Getting started*. URL: https://reactjs.org/docs/getting-started.html (elérés dátuma: 2020. ápr. 30.)

[19] React hivatalos dokumentáció. *Introducing hooks*. 2018. URL: https://reactjs.org/docs/hooks-intro.html (elérés dátuma: 2020. ápr. 25.)

[20] React hivatalos dokumentáció. *Reconciliation*. URL: https://reactjs.org/docs/reconciliation.html (elérés dátuma: 2020. ápr. 30.)

[21] Reaviz hivatalos dokumentáció. *React Router: Declarative Routing for React.js*. URL: https://reaviz.io/?path=/story/docs-intro--page (elérés dátuma: 2020. ápr. 30.)

[22] Scrum hivatalos dokumentáció. *What is Scrum?* URL: https://www.scrum.org/resources/what-is-scrum (elérés dátuma: 2020. máj. 1.)

[23] Semantic UI React hivatalos dokumentáció. *The official Semantic-UI-React integration*. URL: https://react.semantic-ui.com (elérés dátuma: 2020. ápr. 30.)

[24] Semantic UI React hivatalos dokumentáció. *Theming*. URL: https://react.semantic-ui.com/theming (elérés dátuma: 2020. ápr. 30.)

[25] A. Edwards. *An Overview of Go's Tooling*. 2019. ápr. 15. URL: https://www.alexedwards.net/blog/an-overview-of-go-tooling (elérés dátuma: 2020. ápr. 30.)

[26] S. Eskildsen. *Structured, pluggable logging for Go.* URL: https://github.com/sirupsen/logrus (elérés dátuma: 2020. ápr. 30.)

[27] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002, 601–606. oldal.

[28] S. Francia. *Go configuration with fangs*. URL: https://github.com/spf13/viper (elérés dátuma: 2020. ápr. 30.)

[29] A. Ghezala. *You don't have to use Redux*. 2019. jún. 1. URL: https://dev.to/anssamghezala/you-don-t-have-to-use-redux-32a6 (elérés dátuma: 2019. jan. 30.)

[30] Axios hivatalos GitHub oldala. *Promise based HTTP client for the browser and node.js*. URL: https://github.com/axios/axios (elérés dátuma: 2020. ápr. 30.)

[31] MongoDB hivatalos GitHub oldala. *The Go driver for MongoDB*. URL: https://github.com/mongodb/mongo-go-driver (elérés dátuma: 2020. ápr. 25.)

[32] Golangci-lint hivatalos GitHub. *Linters Runner for Go*. URL: https://github.com/golangci/golangci-lint (elérés dátuma: 2020. máj. 1.)

[33] L. Gupta. *What is Golang and how to install it*. 2019. nov. 17. URL: https://medium.com/datadriveninvestor/what-is-golang-and-how-to-install-it-e85002ab0871 (elérés dátuma: 2020. ápr. 30.)

[34] G. Krasner és Stephen Pope. „A cookbook for using the model - view controller user interface paradigm in Smalltalk - 80". *Journal of Object-oriented Programming - JOOP* (1998).

[35] S. Lahoti. *Why Golang is the fastest growing language on GitHub*. 2018. aug. 9. URL: https://hub.packtpub.com/why-golan-is-the-fastest-growing-language-on-github/ (elérés dátuma: 2020. ápr. 30.)

[36] J. Martin. *Managing the Data Base Environment*. Pearson Education, 1983, 381. oldal. URL: https://books.google.ro/books?id=ymy4AAAAIAAJ&pg=PA381&dq=%5C%22CRUD%5C%22.

[37] Q. McCallum. *Bad Data Handbook*. O'Reilly Media, 2012.

[38] J. van Mourik. *GUI for editing your i18n translation files*. URL: https://github.com/jcbvm/i18n-editor (elérés dátuma: 2020. máj. 1.)

[39] D. Pásztor. *UX Design*. UXStudio, Budapest, 2016.

[40] J. Potter. *Compare NPM package downloads*. URL: https://www.npmtrends.com/@angular/core-vs-angular-vs-react-vs-vue-vs-ember-source (elérés dátuma: 2020. ápr. 30.)

[41] T. Rascia. *Build a CRUD App in React with Hooks*. 2018. nov. 7. URL: https://www.taniarascia.com/crud-app-in-react-with-hooks/ (elérés dátuma: 2020. ápr. 30.)

[42] M. Rehkopf. *What is a Kanban Board?* URL: https://www.atlassian.com/agile/kanban/boards (elérés dátuma: 2020. máj. 1.)

[43] I. Sacolick. *What is CI/CD?* 2020. jan. 17. URL: https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html (elérés dátuma: 2020. máj. 1.)

[44] J. Stackhouse. *Cross Origin Resource Sharing middleware for gin-gonic*. URL: https://github.com/itsjamie/gin-cors (elérés dátuma: 2020. ápr. 30.)

[45] Palantir Technologies. *TSLint: An extensible linter for the TypeScript language*. URL: https://github.com/palantir/tslint (elérés dátuma: 2020. máj. 1.)

[46] W. Wattearachchi. *Testing with Jest and Enzyme in React*. hatrészes széria. URL: https://blog.usejournal.com/testing-with-jest-and-enzyme-in-react-part-6-snapshot-testing-in-jest-72fb0ce91c5a (elérés dátuma: 2020. ápr. 30.)

[47] Balsamiq hivatalos weboldal. *Balsamiq. Rapid, effective and fun wireframing software*. URL: https://balsamiq.com/ (elérés dátuma: 2020. ápr. 30.)

[48] GitKraken hivatalos weboldal. *GitKraken Git GUI*. URL: https://www.gitkraken.com/git-client (elérés dátuma: 2020. máj. 1.)

[49] GitLab hivatalos weboldal. *The DevOps lifecycle with GitLab*. URL: https://about.gitlab.com/stages-devops-lifecycle/ (elérés dátuma: 2020. máj. 1.)

[50] Postman hivatalos weboldal. *The Collaboration Platform for API Development*. URL: https://www.postman.com/ (elérés dátuma: 2020. máj. 1.)

[51] TypeScript hivatalos weboldal. *TypeScript - The JavaScript that scales.* URL: https://www.typescriptlang.org/docs/home.html (elérés dátuma: 2020. ápr. 30.)

[52] VS Code hivatalos weboldal. *Visual Studio Code - Code Editing. Redefined*. URL: https://code.visualstudio.com/ (elérés dátuma: 2020. máj. 1.)

[53] R. Wieruch. *How to fetch data with React Hooks?* 2019. márc. 7. URL: https://www.robinwieruch.de/react-hooks-fetch-data (elérés dátuma: 2020. ápr. 30.)